General Purpose Computation on Graphics Processing Units Using OpenCL

POLITECNICO DI TORINO

SCUOLA DI DOTTORATO

Dottorato in

Information and System Engineering – XXV ciclo

Tesi di Dottorato

General Purpose Computation on Graphics Processing Units using OpenCL

Fiaz Gul Khan

Student ID 169507

Tutore

Dr. Bartolomeo Montrucchio (DAUIN)

Prof. Carlo Ragusa (DENERG)

Coordinatore del corso di dottorato

Prof. Pietro Laface

March 2013

Acknowledgements

It is difficult to overstate my gratitude to my supervisor, Dr.Bartolomeo Montrucchio and Prof. Carlo Ragusa, for their expert guidance and support throughout the thesis work. Their suggestions and invaluable ideas provided the platform to the whole thesis work. In spite of their extremely busy schedule, I have always found them accessible for suggestions and discussions. I look at them with great respect for their profound knowledge and relentless pursuit for perfection. Their ever-encouraging attitude and help has been immensely valuable. Throughout my thesis-writing period, they provided encouragement, sound advice, and lots of good ideas. I would have been lost without them.

I wish to convey my sincere thanks to my parents and my family members; who have given me an endless support and love and who also provided me with the opportunity to reach this far with my studies. I would also like to express my thanks to all my colleagues at Politecnico di Torino Italy for their support during this thesis, especially Mr Usman Shahid for letting me to use his GPU for experimental work. This work could not have been accomplished without their support. I also want to express my thanks to Department of Computer and Control Engineering Politecnico di Torino for providing me the logistic support for my work.

Last but not he least special thanks to HEC (Higher Education Commission of Pakistan) for providing me the financial support for my higher educations over here in Italy.

In the end I would like to dedicate this thesis to wonderful parents, especially to my mother, who have raised me to be the person I am today. You have been with me every step of the way, through good times and bad. Thank you for all the unconditional love, guidance, and support that you have always given me, helping me to succeed and instilling in me the confidence that I am capable of doing anything

I put my mind to. Thank you for everything. I love you!

Summary

Computational Science has emerged as a third pillar of science along with theory and experiment, where the parallelization for scientific computing is promised by different shared and distributed memory architectures such as, super-computer systems, grid and cluster based systems, multi-core and multiprocessor systems etc. In the recent years the use of GPUs (Graphic Processing Units) for General purpose computing commonly known as GPGPU made it an exciting addition to high performance computing systems (HPC) with respect to price and performance ratio. Current GPUs consist of several hundred computing cores arranged in streaming multi-processors so the degree of parallelism is promising. Moreover with the development of new and easy to use interfacing tools and programming languages such as OpenCL and CUDA made the GPUs suitable for different computation demanding applications such as micromagnetic simulations.

In micromagnetic simulations, the study of magnetic behavior at very small time and space scale demands a huge computation time, where the calculation of magnetostatic field with complexity of $O(NlogN)$ using FFT algorithm for discrete convolution is the main contribution towards the whole simulation time, and it is computed many times at each time step interval. This study and observation of magnetization behavior at sub-nanosecond time-scales is crucial to a number of areas such as magnetic sensors, non volatile storage devices and magnetic nanowires etc. Since micromagnetic codes in general are suitable for parallel programming as it can be easily divided into independent parts which can run in parallel, therefore current trend for micromagnetic code concerns shifting the computationally intensive parts to GPUs.

My PhD work mainly focuses on the development of highly parallel magnetostatic field solver for micromagnetic simulators on GPUs. I am using OpenCL for

GPU implementation, with consideration that it is an open standard for parallel programming of heterogeneous systems for cross platform. The magnetostatic field calculation is dominated by the multidimensional FFTs (Fast Fourier Transform) computation. Therefore i have developed the specialized OpenCL based 3D-FFT library for magnetostatic field calculation which made it possible to fully exploit the zero padded input data with out transposition and symmetries inherent in the field calculation. Moreover it also provides a common interface for different vendors' GPU. In order to fully utilize the GPUs parallel architecture the code needs to handle many hardware specific technicalities such as coalesced memory access, data transfer overhead between GPU and CPU, GPU global memory utilization, arithmetic computation, batch execution etc.

In the second step to further increase the level of parallelism and performance, I have developed a parallel magnetostatic field solver on multiple GPUs. Utilizing multiple GPUs avoids dealing with many of the limitations of GPUs (e.g., on-chip memory resources) by exploiting the combined resources of multiple on board GPUs. The GPU implementation have shown an impressive speedup against equivalent OpenMp based parallel implementation on CPU, which means the micromagnetic simulations which require weeks of computation on CPU now can be performed very fast in hours or even in minutes on GPUs.

In parallel I also worked on ordered queue management on GPUs. Ordered queue management is used in many applications including real-time systems, operating systems, and discrete event simulations. In most cases, the efficiency of an application itself depends on usage of a sorting algorithm for priority queues. Lately, the usage of graphic cards for general purpose computing has again revisited sorting algorithms. In this work i have presented the analysis of different sorting algorithms with respect to sorting time, sorting rate and speedup on different GPU and CPU architectures and provided a new sorting technique on GPUs.

Contents

Acknowledgements II

Summary IV

I Introduction 1

1 Introduction 2

1.1 Main Contributions 4

1.2 Organization of the thesis 5

2 GPU Computing 7

2.1 High Performance Computing (HPC) 7

2.2 Parallel and Distributed Architectures 8

2.2.1 Multicore shared memory architectures 8

2.2.2 Distributed Memory Architectures 9

2.2.3 Many core Architectures 10

2.3 General-purpose computing on graphics processing units (GPGPU) . 11

2.3.1 Why GPGPU 12

2.3.1.1 Performance 13

2.3.1.2 Architecture 14

2.3.1.3 Memory Bandwidth 15

2.3.1.4 Economical 15

2.3.1.5 Limitations 17

2.4 GPU Programming 18

2.4.1 Open Computing Language (OpenCL) 18

2.4.1.1 Platform Model 19
2.4.1.2 Execution Model 20
2.4.1.3 Memory Model 22
2.4.1.4 Programming Model 23
2.4.2 CUDA 25

II Parallel Magnetostatic solver on GPUs 27

3 Parallel magnetostatic field computation on GPUs using Open Computing Language 28
3.1 GPU Based Parallelization 29
3.1.1 Magnetostatic field computation on GPUs 29
3.2 Micromagnetic Model 31
3.2.1 Micromagnetics 31
3.2.2 Continuum hypothesis 31
3.2.3 Micromagnetic free energy 32
3.2.4 The Dynamic Landau-Lifshitz-Gilbert Equation 33
3.2.5 Effective Magnetic Fields 36
3.3 Magnetostatic Field Computation 38
3.4 FFT discrete convolution method 40
3.4.1 Compute Complexity 43
3.5 Current state of the art of micromagnetic solvers 44
3.5.1 GPU based solvers 45
3.5.2 Limitations of current solvers 46
3.6 OpenCL based GPU implementation 47
3.6.1 Symmetry Properties of Components 47
3.6.1.1 Demagnetizing Tensor 48
3.6.1.2 Magnetization and magnetostatic field 48
3.7 Implementation Approaches 49
3.7.1 GPU-Optimized Implementation 49
3.7.1.1 Minimizing Data Transfer Between CPU and GPU . 50
3.7.1.2 Coalesced Memory Access 51
3.7.1.3 Minimizing GPU memory Utilization 53

3.7.1.4 Minimizing the Arithmetic Computation 54
3.7.1.5 Batch Execution 54
3.7.2 FFTs Based Optimizations 55
3.7.2.1 OpenCL Based 3-D FFT library on GPUs 55
3.7.2.2 Savings in Forward 3-D FFT of Magnetization Vectors 55
3.7.2.3 Savings in Inverse 3-D FFT of Magnetostatic Field Vectors 57
3.7.2.4 Savings in 3-D FFT of Demagnetization Tensor . . . 58
3.8 Performance evaluation 59
3.8.1 Experimental setup 61
3.8.2 Results and discussion 62
3.8.2.1 Computation Time 62
3.8.2.2 SpeedUp 64
3.8.3 Conclusion 67

4 Magnetostatic field computation on multiple GPUs 70
4.1 Why parallel GPUs 70
4.1.1 Limitations of single GPU implementation 71
4.1.2 Multiple GPUs advantages 71
4.2 Different architectural approaches 72
4.2.1 Shared system GPUs 72
4.2.2 Distributed system GPUs 73
4.3 Magnetostatic field computation on multiple GPUs 74
4.3.1 Execution model on multiple GPUs 76
4.3.1.1 Single Context Multiple Devices 76
4.3.1.2 Multiple Contexts Multiple Devices 77
4.3.2 Work division on multiple GPUs 78
4.3.3 Sharing memory objects on multiple GPUs 79
4.3.4 Performance evaluation on multiple GPUs 80
4.3.4.1 Experimental setup 80
4.3.4.2 Computation Time 80
4.3.5 Conclusion 82

III Queue Management on GPUs 84

5 Fast parallel sorting algorithms on GPUs 85

5.1 Introduction . . . 85
5.2 Related Work . . . 86
5.3 Butterfly structure . . . 87
5.3.1 Min-Max Butterfly . . . 87
5.3.2 Full Butterfly Sorting . . . 87
5.4 Performance Analysis . . . 91
5.4.1 Experimental Setup . . . 91
5.4.2 Results . . . 92
5.4.2.1 Sorting Time . . . 92
5.4.2.2 Sorting Rate . . . 93
5.4.2.3 Speedup . . . 95
5.5 Conclusion . . . 97

6 Conclusion 98

Bibliography 101

List of Figures

2.1 Shared Memory CPU Architecture 9
2.2 Distributed Memory Architecture 10
2.3 Nvidia GTX-260 Device Architecture (LM: Local Memory, PE: Processing Element, PM: Private Memory) 11
2.4 New Moore's law (source CMSC828E Spring 2009 lectures) 13
2.5 GPU performance in floating point operations per second against CPU over the years 14
2.6 Architectural difference between CPU and GPU where CPU contains a few high functionality cores while GPU contains 100's of basic cores 15
2.7 Memory Bandwidth GPU vs CPU over the years 16
2.8 OpenCL Programming FrameWork (source opencl overview by khronos group,2011) 19
2.9 OpenCL Platform Model (where PE is Processing Elements) 20
2.10 2-D Addressing Scheme for Work Items or Threads 22
2.11 Hierarchy of OpenCL memory model 23

3.1 Short and long range magnetic interactions 32
3.2 (a)Undamped gyromagnetic precession, (b) Damped gyromagnetic precession 35
3.3 Ferromagnetic body discretization along$X, Y andZ$ direction with number of cell $n_x$along X direction, $n_y$along Y direction and $n_z$along Z direction in the Cartesian coordinates 42
3.4 Demagnetization Tensor $N_{(i-i',j-j',k-k')}$ 43
3.5 2-D magnetization Vector Zero Padding 47
3.6 NVIDIA GTX-260 GPU Architecture 50

3.7 Complex Number Coalesced Memory Access 52
3.8 Memory Coalescing in Transforms 53
3.9 Improvement by changing order of inverse. The plot shows the ratio of improvement for two graphic cards. The presence of the trenches is due to memory coalescing factors. GTX-260, a relatively old card, performs less memory coalescing compared to the Quadro 6000. As a result, the Quadro 6000 appears to be more stable. 59
3.10 Reductions in the demagnetization tensor for the forward transform . 59
3.11 Reductions in the magnetization vector for the forward transform . . 60
3.12 Reductions in the inverse for the magnetization vector using the same 3D FFT routines 60
3.13 Reductions in the inverse for the magnetization vector using a different order of FFT's 61
3.14 Double Precision Computation Times 63
3.15 Single Precision Computation Times 63
3.16 Double Precision GPU/CPU(oommf) SpeedUp where CPU code is running on 4 cores 65
3.17 Single Precision GPU/CPU(oommf) SpeedUp where CPU code is running on 4 cores 65
3.18 Double Precision GPU/CPU(Equivalent CPU Implementation) SpeedUp where CPU code is running on 4 cores 66
3.19 Single Precision GPU/CPU(Equivalent CPU Implementation) SpeedUp where CPU code is running on 4 cores 66

4.1 Shared System Multiple GPUs on multicore CPU system and sharing same CPU memory 73
4.2 Distributed System Multiple GPUs where each CPU node contains one GPU and connected to each other through underlying communication network 74
4.3 The data communication among multiple GPUs using single context multiple devices approach 77
4.4 The data communication among multiple GPUs using multiple context multiple devices approach 77

4.5 Data division among multiple GPUs 79
4.6 Magnetostatic field computation time on single GPU against four GPUs in parallel for different input problem sizes 81
4.7 The speedup of Magnetostatic field computation on four parallel GPUs against single GPU implementation for different input problem sizes 82

5.1 Min-Max Butterfly 89
5.2 Butterfly Sorting 91
5.3 Sorting Time Min-Max Butterfly 92
5.4 Sorting Time Full Butterfly Sort 93
5.5 Sorting Rate Min-Max Butterfly 94
5.6 Sorting Rate Full Butterfly Sort 94
5.7 SpeedUp Full Butterfly Sort Against Bitonic Sort 95
5.8 SpeedUp Full Butterfly Sort Against Different Sorting Algorithms . . 96
5.9 SpeedUp Serial Butterfly Sort Against Different Sorting Algorithms . 96

List of Tables

2.1 Comparison of current state of the art of different GPUs and CPUs with respect to architecture, price performance ratio and memory bandwidth 17
2.2 Different memory regions defined in OpenCL specification and their access types and allocation by the kernel function and the host program 24
2.3 Different terminologies used in OpenCL and CUDA 26

3.1 Architecture details of GPUs and CPU Used 30
3.2 Classification of current Micromagnetic Solvers based on numerical methods, Architectures (Shared Memory/ Distributed Memory/ GPU) and Language/ API 45
3.3 Symmetry Properties of Fourier Transform 48
3.4 Computation Time of OOMMF CPU Implementation and GPU With Single Precision GPU Implementation Against Different Problem Sizes and GPU/CPU 4-cores Speed Up Factor 64
3.5 Computation Time of Our CPU Implementation and GPU With Single Precision GPU Implementation Against Different Problem Sizes and GPU/CPU 4-cores Speed Up Factor 64
3.6 Computation Time of CPU (OOMMF) and GPU With Double Precision GPU Implementation Against Different Problem Sizes and GPU/CPU 4-cores SpeedUp Factor 67
3.7 Computation Time of CPU (Our Implementation) and GPU With Double Precision GPU Implementation Against Different Problem Sizes and GPU/CPU 4-cores SpeedUp Factor 68

4.1 Architecture details of GForce GTX 295 80

Part I

Introduction

Chapter 1

Introduction

With the advent of fast and sophisticated computers and the development in the field of numerical methods helped to solve the small computational problems very fast. Hence it encouraged the computational scientist to move towards the solutions of problems which require excessive amount of computation time. They have developed different solutions in the field of parallel and distributed computing, where the parallelization for scientific computing is promised by different shared and distributed memory architectures such as, super-computer systems, grid and cluster based systems, multi-core and multiprocessor systems etc. However, factors such as heat dissipation, power consumption and small chip sizes limit the number of microprocessors on a single chip, which also affects the number of parallel threads. On the other hand, in case of distributed memory architectures parallel threads can be very large but the performance of overall system heavily depends on the underlying communication network. Secondly such systems can be very expensive with respect to both cost and power consumption.

In this context the use of GPUs (Graphic Processing Units) for General purpose computing commonly known as GPGPU made it an exciting addition to high performance computing systems (HPC) with respect to price and performance ratio. The GPU itself is a many-core processor where dozens of streaming processors with hundreds of cores support thousands of threads running concurrently on single chip. Since the GPU hardware can be classified as SIMT (single instruction, multiple threads), therefore general purpose CPU-bound applications which have significant

data in-dependency are well suited for such devices. Performance evaluation with respect to GFLOPS shows that GPUs outclass its CPU counterparts by manifolds. A high-end Core-I7 Desktop processor (3.46 GHz) can deliver a peak of 55.36 GFlops as compared to Nvidia Quadro 6000 which gives peak performance of 1030 GFlops.

Moreover with the development of new and easy to use interfacing tools and programming languages such as OpenCL and CUDA made the GPUs suitable for different computation demanding applications such as micromagnetic simulations.

Micromagnetism is a generic term used to study the fundamental magnetization processes (the interactions between the magnetic moments) on a microscopic space and time scale. These interactions are managed by different competing short and long range energy terms. All short range components can be calculated using interactions between direct neighbors to a point of interest, resulting in compute complexity of $O(N)$. On the other end, long range components are calculated for each element of interest against all points in the grid. These are essentially convolution operations corresponding to a complexity of $O(N^2)$and is the main contribution towards the total simulation time. By shifting the convolution to be performed in frequency domain, the complexity can be reduced to $O(NlogN)$. The main contributions here are the forward and inverse multidimensional Fourier transforms.

This study and observation of magnetization behavior at sub-nanosecond timescales is crucial to a number of areas such as magnetic sensors, non volatile storage devices and magnetic nanowires etc. Since micromagnetic codes in general are suitable for parallel programming as it can be easily divided into independent parts which can run in parallel, therefore current trend for micromagnetic code concerns shifting the computationally intensive parts to GPUs. All the current micromagnetic solvers on GPU are CUDA based and uses the general-purpose FFT library (cufft) for the computation of magnetostatic field. This limits the current GPU based magnetostatic solver to NVIDIA based hardware only. Secondly by the use of general-purpose FFT library they cannot fully exploit the zero padded input data without transposition and symmetries inherent in the field calculation. Their design approach limits them to certain size of input problems depending on the global memory of GPU being used.

1.1 Main Contributions

The main goal of this thesis is to develop the highly optimized parallel magnetostatic field solver on GPUs by exploiting the symmetries inherited in the field calculation using specialized multidimensional FFT library on GPUs. I have used OpenCL on GPUs, with consideration that it is an open standard for parallel programming of heterogeneous systems for cross platform. It targets different devices such as GPUs by different vendors such as Nvidia, ATI and Intel etc, along with CPU and other processing hardware which conform to its specification. The magnetostatic field calculation is dominated by the multidimensional FFTs (Fast Fourier Transform) computation. Therefore I have developed the specialized OpenCL based 3D-FFT library for magnetostatic field calculation which made it possible to fully exploit the zero padded input data without transposition and symmetries inherent in the field calculation. As a result the complexity of overall system reduced significantly compared to current GPU based solvers. Moreover it also provides a common interface for different vendors' GPUs. In order to fully utilize the GPUs parallel architecture my solver handles many hardware specific technicalities such as coalesced memory access, data transfer overhead between GPU and CPU, GPU global memory utilization, arithmetic computation, batch execution etc. For the accuracy and performance evaluation I have compared the results with the CPU-based parallel OOMMF program developed at NIST and with an equivalent parallel implementation on CPU and shown an impressive speedup.

In the second step to further increase the level of parallelism and performance and to avoid the limited memory resources on single GPU, I have developed a parallel magnetostatic field solver on multiple GPUs. Utilizing multiple GPUs avoids dealing with many of the limitations of GPUs (e.g., on-chip memory resources) by exploiting the combined resources of multiple on board GPUs. I have shown the implementation of magnetostatic field solver on multiple GPUs and the speedup against single GPU implementation.

In parallel I also worked on ordered queue management on GPUs. Ordered queue management is used in many applications including real-time systems, operating systems, and discrete event simulations. In most cases, the efficiency of an application itself depends on usage of a sorting algorithm for priority queues. Lately,

the usage of graphic cards for general purpose computing has again revisited sorting algorithms. In this work I have presented the analysis of different sorting algorithms with respect to sorting time, sorting rate and speedup on different GPU and CPU architectures and provided a new sorting technique on GPUs.

In short the most relevant outcomes of my work are

- Highly optimized OpenCL based multidimensional FFT library for magnetostatic filed calculation.
- OpenCL based magnetostatic field solver on multiple GPUS.
- Saving the memory and computation time by avoiding the transposition overhead in multidimensional FFT on GPUs.
- Reducing the complexity of both forward and inverse multidimensional FFT on GPU by considering the symmetries and zero padded input data in magnetostatic field solver.
- Reducing the memory transactions on GPU by coalesced memory access.
- Reduction in the data transfer overhead between CPU and GPU by utilizing the symmetries in the input data, with the development of specialized FFT library for magnetostatic field calculation on GPUs.
- Developed a CPU based parallel magnetostatic field solver for comparison with GPU results.
- Implementation and comparison of different sorting algorithms on different GPUs architecture.

1.2 Organization of the thesis

Rest of the thesis is organized as follows. In chapter 2 I have described the emerging importance of GPU computing in the field of high performance computing. I have discussed the importance of GPUs for general purpose computing and the programming methodology for general purpose application on GPUs.

In chapter 3 I have discussed the parallel implementation of micromagnetic simulation on GPUs architecture. I have discussed the micromagnetic model with respect to computational complexity. Then I have magnetostatic field calculation which is the most time and memory consuming part of magnetostatic field solvers. As the computation of magnetostatic field is dominated by the FFTs computation therefore I have developed the specialized multidimensional FFT library on GPUs using OpenCL which fully exploits the zero padded input data and the symmetries inherited in the input data.

In chapter 4 in order to further increase the performance and to over come the limited memory resources on single GPUs I have discussed the parallel implementation of magnetostatic field solver on multiple GPUs. I have discussed the different architectural approaches to use multiple GPUs along with their pros and cons. I have discussed the load division on multiple GPUs and the communication overhead while working on multiple GPUs along with the savings in the computation time.

Sorting algorithms have been studied extensively since past three decades. Their uses are found in many applications including real-time systems, operating systems, and discrete event simulations. In most cases, the efficiency of an application itself depends on usage of a sorting algorithm. Lately, the usage of graphic cards for general purpose computing has again revisited sorting algorithms. In chapter 5 I have presented a novel Butterfly Network Sorting algorithm (BNS) for sorting large data sets on GPUs. A minimal version of the algorithm Min-Max Butterfly is also shown for searching minimum and maximum values in data. Both algorithms are implemented on GPUs using OpenCL exploiting data parallelism model and their results are compared to different serial and parallel sorting algorithms on CPUs and GPUs respectively.

At the end in chapter 6 I have provided the conclusion of my PhD work and the expansion of my work in the future.

Chapter 2

GPU Computing

2.1 High Performance Computing (HPC)

High-performance computing (HPC) is the fast, efficient and reliable execution of computational intensive problems with different parallel processing techniques. Parallelism is the future of computational science, which has emerged as a third pillar of science along with theory and experiment.

Parallelization for scientific computing is promised by architectures such as, super-computer systems, grid computers [1, 2], cluster based systems, and different shared memory architectures. In recent years multi-core and multi-processor computers have become very common [3]. However, factors such as heat dissipation, power consumption and small chip sizes limits the number of microprocessors on a single chip, which also affects the number of parallel threads. On the other hand, in case of distributed memory architectures parallel threads can be very large but the performance of overall system heavily depends on the underlying communication network. Secondly such systems can be very expensive with respect to both cost and power consumption and scalability can also be the issue. In this context the use of GPUs (Graphic Processing Units) [4, 5, 6] for General purpose computing made it an exciting addition to high performance computing systems with respect to price and performance ratio[7, 8].

2.2 Parallel and Distributed Architectures

Flynn's taxonomy classifies serial, parallel and distributed computers architecture according to how the instruction is being executed on available processors. Like in SIMD (single instruction/multiple data) executes the same instructions on all available processors at same time, or each processor executes different instructions like in MIMD (multiple instruction/multiple data). The second way to distinguish between these mode of computing is how different processor communicate among each other. Distributed memory machines communicate by explicit message passing, using tools such as MPI, while shared memory machines have a global memory address space, and tools such as OpenMP can be used to read and write the global memory by the various processors. Beside these CPU based parallel and distributed computer architectures with the advent of new hardware architecture such as graphics processing units (GPUs) provided exciting opportunities in the field of parallel and distributed computing. Each node in such systems comprises hundreds or even thousands of high-performance stream processors. We can summarize these architecture like

- Multicore shared memory architectures.
- Distributed memory architectures.
- Many core architectures.

2.2.1 Multicore shared memory architectures

In recent years multi-core and multi-processor computers have become very common. Multi-core processor have two or more cores on a single chip with their own level-1 cache and can run multiple tasks simultaneously by sharing the common global memory, decreasing the computation time of parallel program as depicted in figure 2.1. The programming languages for multi-core CPUs ranges from low-level multi-tasking or multi-threading libraries like POSIX (pThreads), to high level libraries such as Intel Threading Building Block (TBB).

However, factors such as heat dissipation, power consumption and small chip sizes limits the number of microprocessors on a single chip, which also affects the number of parallel threads. The communication cost in case of multicore processors

is much less as they share the same memory through system bus as compare to distributed memory architecture where communication between nodes takes place over an underlying network.

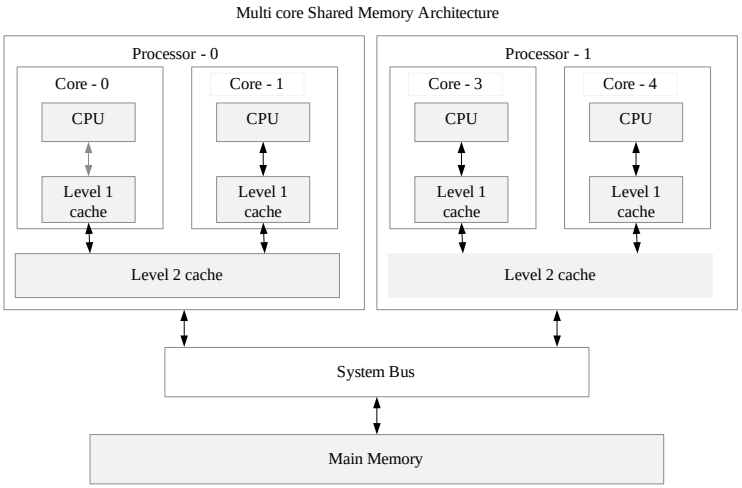

Figure 2.1. Shared Memory CPU Architecture

2.2.2 Distributed Memory Architectures

A distributed memory architecture also know as a message passing multiprocessor or multi-computer as it connects computers which have their own private memory together via underlying communication network as shown in figure 2.2. It becomes indispensable to move on to distributed memory systems such as clusters and super computers when both memory and time becomes problematic on shared memory systems.

There are number of issues while programming for distributed memory systems. The most prominent one is how to distribute the data over the memories as the cost to send data on communication network is much high, so in designing a parallel algorithm for distributed system one must have to divide the data efficiently, otherwise it can degrade the performance of your problem.

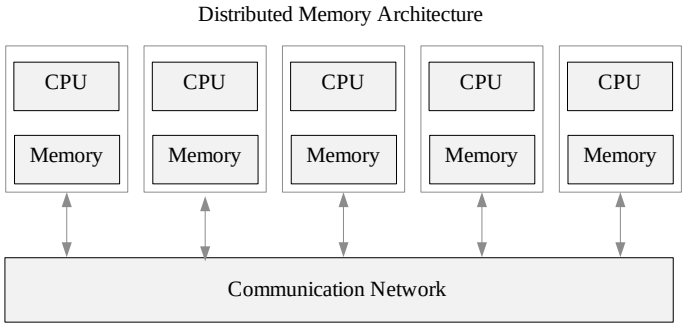

Figure 2.2. Distributed Memory Architecture

2.2.3 Many core Architectures

The graphics processing unit (GPU), which initially was designed for manipulating computer Graphics, now with the development of high level libraries and easy to use interfacing tools such as OpenCL and CUDA can be used as co-processor to speed up wide range of computation intensive applications. The GPU in particular is a many-core processor with support for thousands of concurrently running threads[9]. This is made possible through a particular alignment of dozens of streaming processors including hundreds of cores as shown in Figure2.3. Thread management at hardware level implies that context-switching time is close to none. In addition to high-end games, general purpose applications which are heavily CPU-bound or which have considerable data in-dependency are ideally suited for these devices. Data parallel codes are particularly suited as the hardware can be classified as SIMT (single-instruction, multiple threads). Performance evaluation with respect to GFLOPS shows that GPUs outclass its CPU counterparts by manifolds. A high-end Core-I7 Desktop processor (3.46 GHz) can deliver a peak of 55.36 GFlops as compared to Nvidia Quadro 6000 which gives peak performance of 1030 GFlops.

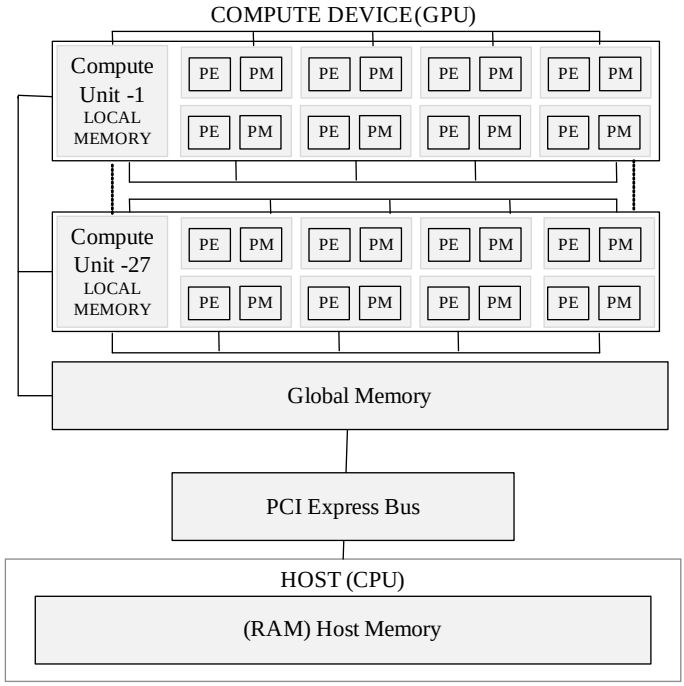

Figure 2.3. Nvidia GTX-260 Device Architecture (LM: Local Memory, PE: Processing Element, PM: Private Memory)

2.3 General-purpose computing on graphics processing units (GPGPU)

GPGPU stands for General-Purpose computation on Graphics Processing Units, some times also referred as GPU Computing. The name *GPGPU* (General Purpose computation Graphic Processing Units) [10, 11, 12] was first coined by M.J Harris in 2002 [13]. The GPU itself is a many-core processor where dozens of streaming processors with hundreds of cores support thousands of threads [9] running concurrently on single chip as depicted in figure 2.3. Since the GPU hardware can be classified as SIMT (single-instruction, multiple threads), therefore general purpose CPU-bound applications which have significant data in-dependency are well suited for such devices. Secondly with the development of high level libraries and easy to use interfacing tools such as OpenCL and CUDA made it easy to use GPU as co-processor to speed up wide range of applications in different areas of scientific computing such as

- Computational Physics
- Image Processing
- Computational Modeling
- Computational Biology
- Computational Geo Science
- Computational Chemistry and many more

2.3.1 Why GPGPU

When we talk about general purpose computation on graphic processing units the first question that most frequently arises is that why GPU is so faster than CPU. In this section I would briefly discuss the motivation behind the use of GPUs for general purpose computation, which were initially designed specifically for graphic applications. According to new Moore's law "computers no longer get faster but just wider so you must be rethinking of your algorithms to be parallel". The current trend of microprocessor industry also endorses the new Moore's law by adding number of cores instead of increasing the clock frequency of single core. According to Herb Sutter of Microsoft in Dr. Dobbs' Journal says "The free lunch is over, software performance will no longer increase from one generation to the next as hardware improves unless it is parallel software". Therefore Parallelism is the future of computational science. The performance of parallel software is heavily dependent on the hardware architecture which you are using for your application. The reasons behind the use of GPUs for general purpose computation are many folds here are some of the most important factors for using GPUs for general purpose applications are in the following sections.

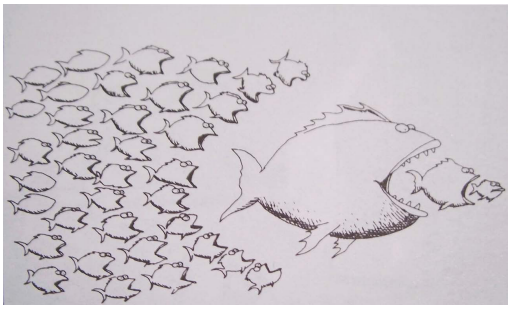

Figure 2.4. New Moore's law (source CMSC828E Spring 2009 lectures)

2.3.1.1 Performance

Computer performance can be measured with FLOPS (i.e. floating point operations per second) specially in the case of computational science where different applications heavily use floating point calculations in their computation. Performance evaluation with respect to GFLOPS shows that GPUs outclass its CPU counterparts by manifolds. For example A high-end Intel Core i7 3960-X (3.30 GHz) can deliver a peak of 141.09 GFlops as compared to ATI Radeon HD 6990 which gives peak performance of 5099 GFlops which is more than 36x faster than its CPU counterpart. Figure 2.5 shows the performance in floating point operations per second of GPUs by different vendors like Nvidia and ATI against CPU over the years. Secondly with respect to speed evolution since the 1990s [14] the GPU performance on average doubled every six months compare to its counterpart CPU whose performance according to Moore's law approximately doubles every eighteen months. This trend is expected to continue in case of GPU technology but on the other hand over the last few years clock frequency of CPU is not getting faster. With respect to power efficiency the current GPUs over the last few years grows even faster than the Moore's law and are most efficient as compared to their counterpart CPUs, for example the ATI's Radeon HD 5870 gives 14.47 GFLOPs/Watt compared to Intel's Core i7-3960X Processor Extreme Edition which gives roughly 1.17 GFLOPs/Watt.

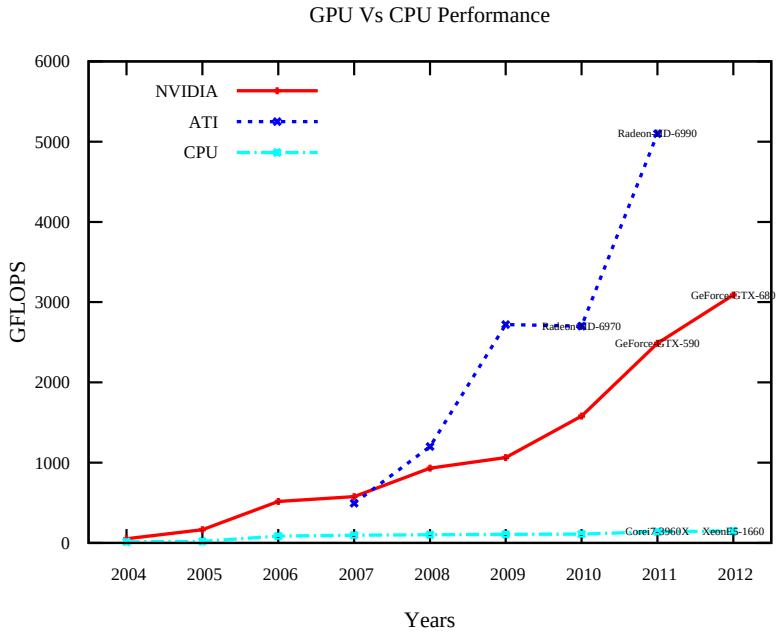

Figure 2.5. GPU performance in floating point operations per second against CPU over the years

2.3.1.2 Architecture

Now the question arises that why GPUs are so fast as compared to CPU. The simple answer to this question is the difference in the architecture of both GPU and CPU. CPU is designed for different kinds of input data especially in consideration with the sequential data. Therefore on CPU a large number of transistors are involved in performing the non computational tasks such as branch prediction, scheduling operations and data transfer etc which limits the number of transistors for computational tasks. On the other hand in case of GPU, it is specially designed for applications with high level of data parallelism and performs the closely defined tasks so there is no communication between the data elements running on the separate processing units of GPU. In this way with the same number of transistors GPU outperforms it counterpart CPU by assigning more number of transistors for computation rather than other purposes as in CPU. Figure shows the difference between the architecture of both GPU and CPU.

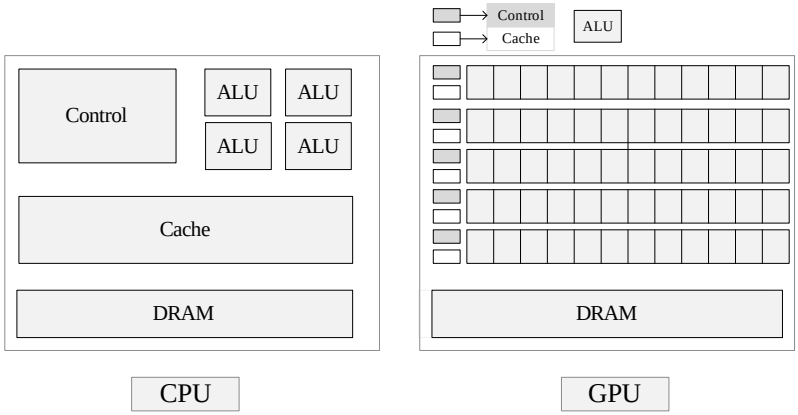

Figure 2.6. Architectural difference between CPU and GPU where CPU contains a few high functionality cores while GPU contains 100's of basic cores

2.3.1.3 Memory Bandwidth

The performance of an application on GPU mainly depends on the memory bandwidth. Infect the most important factor for the optimization of a code on GPUs is the effective use of the memory bandwidth. GPU have different memory hierarchy and the location of your data on different hierarchy of memory and the way you access this data effects the overall performance of an application dramatically. The detailed description of these different memory hierarchies and the methods to access the data from these memories are explained in detailed in the respective sections. Most of the recent GPUs have their own dedicated Global memory with high bandwidth like for example ATI's Radeon HD 7970 have 288 GBps global memory bandwidth. Figure 2.7 show the comparison of memory bandwidth between different high end CPUs and GPUs over the years. From figure 2.7 we can see a huge difference between the bandwidth of CPU and GPU and this gap is constantly increasing over the years, this is made possible through large memory bus width on GPU.

2.3.1.4 Economical

The high computational power provided by the GPU is also inexpensive with respect to CPU. Graphic Processing Units came out as an alternative for parallel computing with respect to price and performance ratio compared to current shared

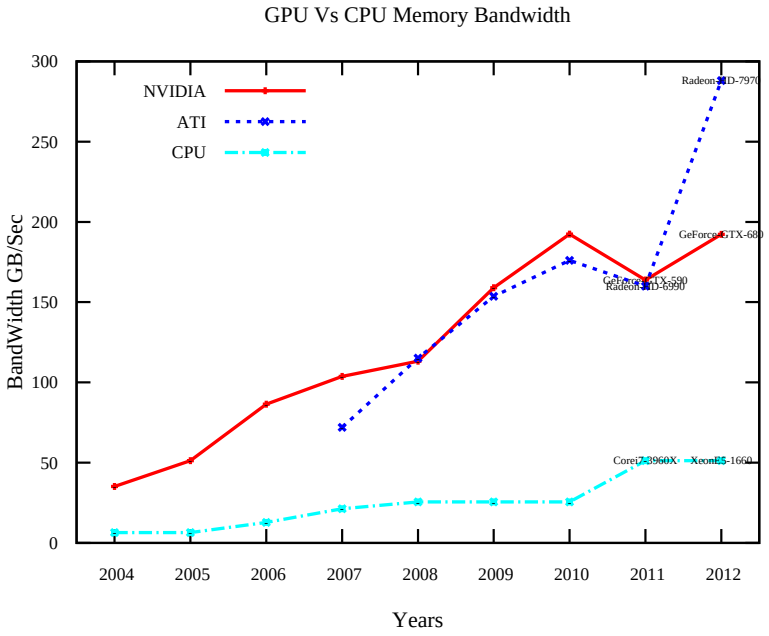

Figure 2.7. Memory Bandwidth GPU vs CPU over the years

and distributed memory systems. The average cost of typical GPU when it releases is between four hundred to six hundred dollars. Comparing GFLOPS per dollar, the Core i7 980X costs $999 and gets roughly 0.1 GFLOPS/$, whereas the HD 5970 costs $599 and gets 1.5 GFLOPS/$ at double precision and 7.7 GFLOPS/$ at single precision. This high performance at very low cast has also force the high performance computing industry to the use GPUs as an accelerator. In the statistics published for top 500 supercomputers on November 2011, thirty nine systems were using GPUs as an accelerator which was increased from seventeen just in six months and this trend is expected to continue more rapidly in the future.

Table 2.1 Summarizes the current state of the art of different GPUs by Nvidia and ATI and their counterpart CPU with respect to different parameters such as number of cores, memory bandwidth and price performance ratio. In all fields GPUs outclass its CPU counterparts by manifolds.

Specification	NVIDIA Cards		ATI Cards		Intel CPU	
	GeForce GTX 680	GeForce GTX 590	Radeon HD 7970	Radeon HD 6990	Intel Xeon E5-1660	Core i7 3960X
Release date	March 22, 2012	March 24, 2011	Jun 22, 2012	March 8, 2011	Quarter 1, 2012	Quater 1, 2011
Total Cores	1536	2*512	2048	2*1536	6	6
GFLOPS	3090.4	2488.3	4300	5099	149.16	141.09
Price in $	499	699	499	699	1080	1059
GFLOPS/$	6.19	3.56	8.62	7.3	0.13	0.133
Mem. Bandwidth (GB/s)	192.256	163.87	288	160	51.2	51.2

Table 2.1. Comparison of current state of the art of different GPUs and CPUs with respect to architecture, price performance ratio and memory bandwidth

2.3.1.5 Limitations

Despite the huge raw computational power of GPU for general purpose applications and the support for new easy to use programming languages and interfaces it has still many limitations As the GPU hardware is classified as SIMT therefore it is still limited to certain class of parallel applications where there is no or very little data dependency. Secondly GPU programming is not just about learning a new programming language, in order to fully utilize the GPUs parallel architecture the programmer must understand the underlying hardware design and have to modify problem accordingly. Moreover the latency in the data transfer between GPU and CPU is another big issue. The applications which involve lot of data communication between CPU and GPU even running on GPU do not show any improvement or even get worst performance. Therefore it is the single most important consideration while designing an algorithm for GPU. The support for double precision floating point accuracy is another issue on GPUs. Most of the current GPUs lack the support for double precision while the others sacrifices IEEE compliance. In scientific computing accuracy of the results many applications is of primary importance. Therefore the lack or limited support of double precision accuracy on GPUs limits its usage for many scientific problems. As the GPUs technology is rapidly changing therefore there is no well defined standard for GPU computing. In this regards by collective effort of many different companies like Nvidia, ATI, Intel etc., OpenCL came out as remedy but is still in its developing stages.

But despite all these challenges, the potential benefits as discussed above and the growth curve with respect to its counterpart CPUs it is hard to ignore to use it for general purpose applications.

2.4 GPU Programming

In the recent years Graphic Processing Units (GPUs) emerged as a strong candidate for parallel computing with respect to price performance ratio compared to current shared and distributed memory systems. As a result of collective efforts by industry and academia, modular and specialized hardware in the form of sound cards or graphic accelerators now can be used for general purpose applications. Earlier credits to NVIDIA by adding programmable graphics pipeline to GPUs and AMD/ATI introducing floating point math capability, has led GPUs for general purpose computation. Recent developments in dedicated parallel programming model and APIs like NVIDIA-CUDA [15] and OpenCL specification [16] by Khronos Group enabled GPUs to offload CPU burden for fast numerical crunching working as co-processor [17]. In this section I would briefly explain both the CUDA and OpenCL and the key difference between these two languages.

2.4.1 Open Computing Language (OpenCL)

In the recent years different vendors provided GPU programming API's such as CUDA by NVIDIA and ATI's FireStream . The GPU code developed using these APIs is usually not portable among GPU devices developed by different vendors. Secondly the scalability of parallel processor on a single chip is also becoming a great challenge due to many different reasons like, space on chip, heat dissipation, power consumption etc. Therefore in order to increase the parallel performance one have to utilize all the available resources present on a system such as GPU, CPU and other processing architecture present on a system. Open Computing Language (OpenCL) by Khronos group seems to be remedy for these challenges.

OpenCL is a standard for general purpose parallel programming for heterogeneous processors. It allows the development of parallel code that takes advantage of computing power of different computing devices present on a system such as

CPU, GPU, cell broadband engine and other processing devices which conform to its specification[18, 19]. OpenCL is based on ISO C99 with some extensions for parallel programming and can support both task and data based parallel programming models.

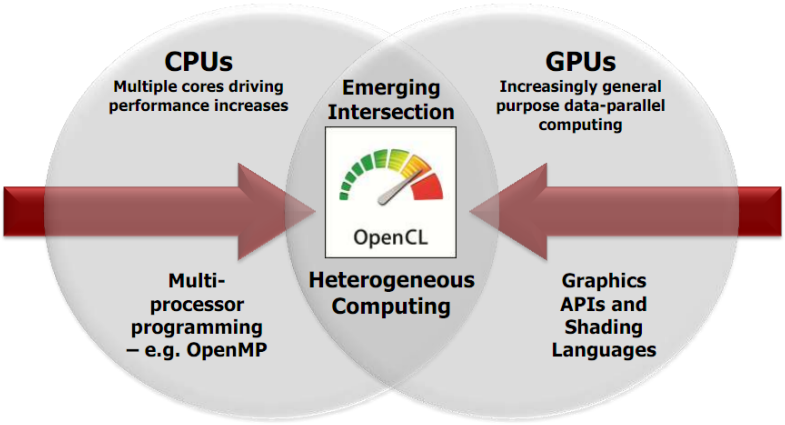

Figure 2.8. OpenCL Programming FrameWork (source opencl overview by khronos group,2011)

The working of OpenCL can be explained by using the following models' hierarchy

- Platform Model
- Execution Model
- Memory Model
- Programming Model

2.4.1.1 Platform Model

The OpenCL environment is defined by the platform, where there can be different devices which are controlled from platform. These devices are managed by creating the context where each device can have its own separate context or different devices

may have one single context. Inside each context different tasks are scheduled to execute on OpenCL devices through command queue.

The OpenCL platform model consists of two parts that are host and device. In OpenCL the processor which executes the main code is referred as host while the processor which run the kernel written in OpenCL programming language is referred as device. One or more devices which conform to OpenCL specification are connected to Host. The device is comprises of different compute units like one of the GPU device which I am using that is Nvidia GTX-260 has twenty seven compute units and each compute unit have different processing elements which are eight in case of GTX-260. The application running on the host side controls the execution of commands on device side. The different components of OpenCL platform are shown in figure 2.9

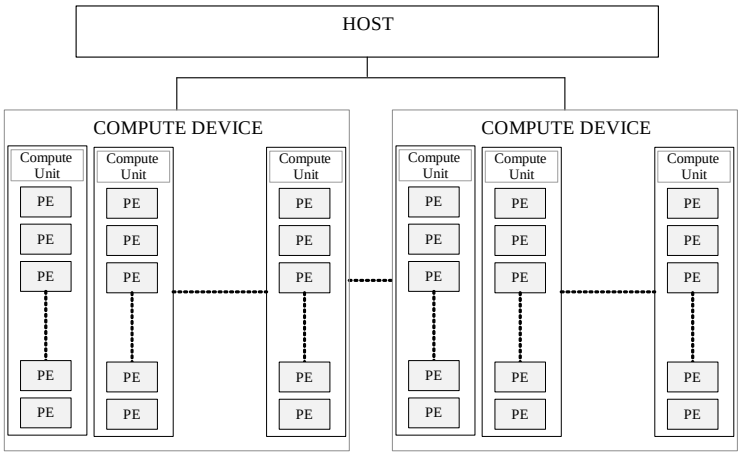

Figure 2.9. OpenCL Platform Model (where PE is Processing Elements)

2.4.1.2 Execution Model

The way in which the host program is to be executed and executes the kernel on the OpenCL device is defined by the execution model. The OpenCL program executes in two parts

1. Kernel program

2. Host program

The kernel program, also called device program is a specific piece of code running on device (s) and is executed concurrently by several threads and this is where data/task parallelism takes place. The other, called a host program, runs entirely on host side that launches kernels i.e. SIMT based programs and manages the execution of kernel on device side.

The host program manages the whole problem size to be executed on device by creating the index space. The indexed space which is in OpenCL called ND-Range can be an N dimensional ND-Range, where N can be 1, 2 or 3. The host program when submits the kernel to be executed on the device side with defined ND-Range; the device executes the kernel for each point of ND-Range. Each instance of kernel in the ND-Range is known as work item. Work items are divided into work groups. A work item is the basic execution unit in the ND-Range.

Each work-item or thread within the NDRange is identified by a global and local addressing scheme, both of which are based on the dimensional sizes of NDRange and its work-groups. Work items inside a work-group are addressed by local addressing with scope only to current work-group. Work items belonging to different work-groups can have same local addressing but not global one. This scheme is outlined in figure 2.10 for a two dimensional problem. A single dimensional address can be computed as

$$global_{id} = (workgroup_{id} * workgroup_{size})$$

provided that it fulfills the following condition

$$0 \leq local_{id} \leq workgroup_{size}$$

and ND-Range can be calculated by the following expression

$$ND - Range = max(workgroup_{id}) * workgroup_{size}$$

$$\frac{ND - Range}{max(workgroup_{id})} \% 2 = 0$$

here the $workgroup_{size}$ and the ND-Range is defined by the programmer.

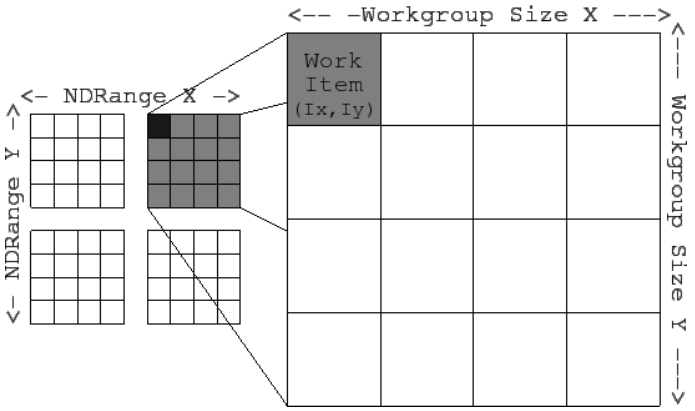

Figure 2.10. 2-D Addressing Scheme for Work Items or Threads

2.4.1.3 Memory Model

OpenCL memory model defines four different types of memory that a kernel function running on OpenCL device can access. The memory model also specifies the access to these memory regions. These four different memory regions are:

Global Memory On OpenCL device global memory region is the largest memory region and is accessible to all threads running on the device. It provides both read and write access to all work items running on the OpenCL device. Among four different types of memory region the read / write access to global memory is considered to be the slowest. In order to achieve the best performance the global memory must be access in a coalesced way to exploit the full memory bandwidth. The coalesced memory access would be discussed in detail in the later on sections.

Constant Memory Constant memory is the read only section of the global memory for the kernel and the data on constant memory remains constant during the kernel's execution. Constant memory region is considered to be good for broadcast data.

Local Memory Local memory is generally an on chip memory and is faster than global memory. As shown from its name it is local to work-group running on a

compute device as shown in figure 2.11.

Private Memory The concept of private memory is similar to register on CPU. It is the faster among all four memory region and is private to the work item running on specific processing element. The work items running on different compute units have no access to private memory of other compute units.

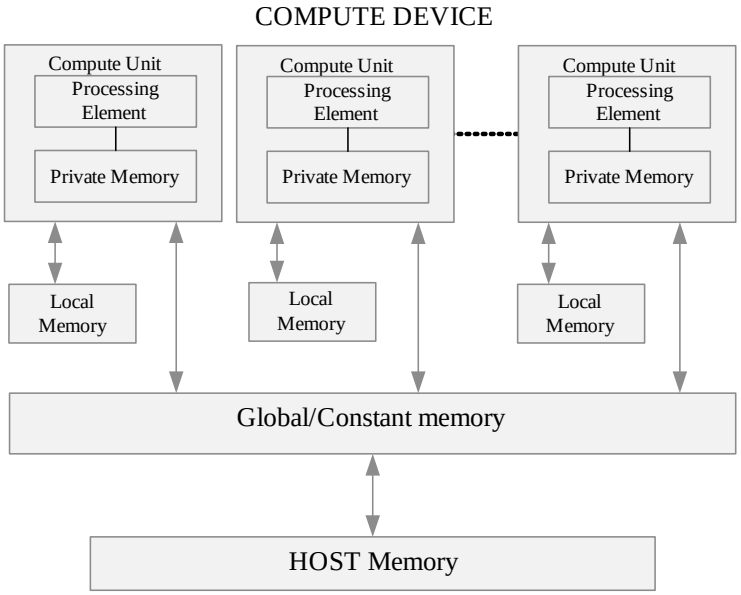

Figure 2.11. Hierarchy of OpenCL memory model

2.4.1.4 Programming Model

As OpenCL targets different processing hardware which conform to its specification such as GPU and CPU, therefore it supports both data and task parallel programming model. Where task parallel execution mode enables it to use CPU. On the other hand as CUDA supports only GPUs therefore it is not available in CUDA.

Type of Memory	Kernel function		Host Program	
	Allocation Type	Access	Allocation Type	Access
Global Memory	No Allocation	Read / Write access	Dynamic Allocation	Read / Write access
Constant Memory	Static Allocation	Read only access	Dynamic Allocation	Read / Write access
Local Memory	Static Allocation	Read / Write access	Dynamic Allocation	No Access
Private Memory	Static Allocation	Read / Write access	No Allocation	No Access

Table 2.2. Different memory regions defined in OpenCL specification and their access types and allocation by the kernel function and the host program

Data Parallel Programming Model In data parallel programming model different threads follows same instruction on different elements of data. In OpenCL threads are organized in ND-Range and these threads are mapped to the data to be processed. The data mapping to threads can be of two types that is strict or relaxed data mapping. The strict data mapping follows the one to one mapping of threads and the data elements to be processed, while on the other hand like in OpenCL where there is relaxed version of data mapping in which one to one mapping between the threads and the data elements is not required.

Along with relaxed data parallel programming model OpenCL data parallel programming model is also hierarchical, which means at first level divide the data to be processed among the threads or work items and at second level these work items are organized into work groups to execute in parallel. The division of work items into work-groups can be either implicit or explicit. In implicit division the programmer only specifies the total number of work items and the OpenCL implementation manages the division of these work items into work-groups. While in explicit parallel model both the number of work items and the size of work group is defined by the programmer. The choice of hierarchical model depends on the nature of the application.

Task Parallel Programming Model In task parallel programming model generally different threads follows different instructions on same or different data elements. In OpenCL instructions are executed in the form of kernel function and in

task parallel model for each task single instance of kernel executes and parallelism can be achieved by executing multiple kernels for different instructions. In other words we can say that each work group on a compute unit contains only one work item.

Synchronization In OpenCL the synchronization among the parallel execution of threads can be achieved at two different levels

1. Work-group level

2. Command Queue level

Work-group level synchronization The synchronization among different work items belonging to same work group is achieved by using work group barrier. The work group barrier ensures that all the work items in a work group would not proceed further before each work item in a work group reached that point of execution. On the other hand we can not perform synchronization among different work items belonging to different work-groups in an OpenCL application.

Command-Queue level synchronization The second level of synchronization in OpenCL is at command queue level, where we can use "command queue barrier" or "clWaitForEvents" to perform synchronization among different commands belonging to a same command queue.

2.4.2 CUDA

CUDA (Compute Unified Device Architecture) is a platform and programming model for parallel computing on CUDA enabled devices by NVIDIA. As the CUDA architecture is similar to the OpenCL architecture explained in section 2.4.1 [20] that's why I would not go into its detailed architecture. Here in this section I would only explain one of the some major differences between the OpenCL and the CUDA that is Homogeneous vs Heterogeneous and the table 2.3 provides the difference between the terminologies used in both platforms.

Homogeneous vs Heterogeneous The GPU is programmable using different GPU computing platforms such as NVIDIA's CUDA and OpenCL by Khronos group. Where CUDA only targets GPU devices by Nvidia (homogeneous), OpenCL targets different devices such as GPUs by different vendors such as Nvidia, ATI and Intel etc, along with CPU and other processing hardware which conform to its specification (very heterogeneous). Despite the reason that CUDA is more developed and matured than OpenCL, heterogeneous property of OpenCL make it superior and more futuristic and is forcing the developers to choose OpenCL.

There is a famous quote by Senior Mathematician Jeff Lait at Side Effects Software talk that is *"Volume simulation like Houdini's PyroFX involves highly parallel operations on large datasets: exactly what GPUs are best at. Our original tests were performed in CUDA, but as we are not in the position to dictate the hardware used by our customers, we wanted the final version to be as hardware-agnostic as possible. OpenCL fitted this requirement."*

OpenCL Terminologies	**CUDA Equivalent**
Device	GPU
Compute Unit	Multiprocessor
Processing Element	Scalar Core
Global Memory	Global Memory
Local Memory	Shared Memory
Private Memory	Local Memory
ND-Range	Grid
Work-Group	Block
Work-Item	Thread

Table 2.3. Different terminologies used in OpenCL and CUDA

Part II

Parallel Magnetostatic solver on GPUs

Chapter 3

Parallel magnetostatic field computation on GPUs using Open Computing Language

Recent Graphic Processing Units (GPUs) have remarkable raw computing power, which can be utilized for high computational challenging problems. This is the case of micromagnetic simulations, where the study of magnetic behavior at very small time and space scale demands a huge computation time. Here the calculation of magnetostatic field with complexity of $O(NlogN)$ using FFT algorithm for discrete convolution is the main contribution towards the whole simulation time. In this chapter I will present a magnetostatic field solver for micromagnetic simulators on GPUs. For my implementation I am using OpenCL for GPU implementation, with consideration that it is an open standard for parallel programming of heterogeneous systems for cross platform. Secondly, I have developed a specialized OpenCL based 3D-FFT library for magnetostatic field calculation made it possible to fully exploit the symmetries inherent in the field calculation and other optimizations specific to GPU architecture. I have implemented this magnetostatic field solver for both single and double precision floating point accuracy on different GPU architectures. For the accuracy and performance evaluation I compared my results with the CPU-based parallel OOMMF program developed at NIST and with an equivalent parallel implementation on CPU. I find out a speedup of up to 94x for single and 45x

for double precision floating point accuracy against my equivalent OpenMp based parallel CPU implementation. Against OOMMF I am getting the speedup of up to 8.6x for single and 4.4x for double precision floating point accuracy.

3.1 GPU Based Parallelization

Chip level parallelism is a driving force in recent advancements in microprocessor architectures, as a result of which multi-core CPUs [3] are commonly available in the market. But since most of today's personal computers are used for gaming and entertainment purpose, the core processors were not sufficient. As a result, modular and specialized hardware in the form of sound cards or graphic accelerators are increasingly present in most personal computers. These devices provide better experience as compared to traditional on-board mechanisms. Over the years, they have improved with respect to sophistication and recently, graphics cards or graphics processing units (GPU) in addition to high-end gaming can also be used as a co-processor to the CPU for general purpose computing.

3.1.1 Magnetostatic field computation on GPUs

The emergence of GPUs for general purpose computing, opened the gates for researchers to use GPUs for wide range of computational challenging problems such as micromagnetic simulations. Applications involving massive data-parallelism are ideally fitted to the GPU architecture. Since micromagnetic codes in general are suitable for parallel programming as it can be easily divided in to independent parts which can run in parallel [21], therefore current trend for micromagnetic code concerns shifting the computationally intensive parts to GPUs. The magnetostatic field computation is the most time and memory consuming part of the simulation [22, 23] and is iteratively obtained for each time interval. The magnetostatic field at a given point is calculated by taking considerations from the complete magnetization vector field, which involves interactions performed over a long range. This results in an asymptotic complexity of $O(n^2)$ where n is the number of field points. For regular grids, the calculations correspond to a convolution operation, and thus by shifting the convolution to be performed in the frequency domain, the complexity can be

reduced to $O(n \log n)$. This is the standard approach taken by most current simulators, which make use of various general purpose FFT libraries to perform the transformations. The transformation process is the main contributor to the field calculation time.

In this chapter I would discuss the highly optimized OpenCL based magnetostatic field solver on GPU. Hardware vendors usually provide a set of high-performance general purpose FFT routines optimized for their hardware like clAmdFft by AMD, clFFT by Apple, and cufft by Nvidia and their interfaces to FFT routines are also different. Therefore in my implementation for performing the transforms, I have developed my own specialized OpenCL based 3-D FFT library for magnetostatic field solvers on GPUs. The OpenCL based 3-D FFT library for GPUs provides us the freedom to fully exploit the symmetric and zero padded input data, optimizations specific to GPU hardware and also provides a common interface for different vendors' GPUs.

Table-1 reports some architecture details of GPUs and Intel Core2 Quad system that I have used for my implementation of magnetostatic field solver. The devices include a high-end graphics card like Quadro 6000 comprising of 14 stream processors with 32 cores each, and NVIDIA GTX 260 with 27 processors having 8 cores each [24].

Architecture Details	**NVIDIA**		**Intel**
	Quadro 6000	**GTX 260**	**Core2 Quad Q8400**
Total Cores	448	216	4
Micro Processors	14	27	1
Clock Rate (MHz)	574	576	2660
GFLOPS	1030.4	874.8	42.56
Mem. Bandwidth (GB/s)	144	91.36	-

Table 3.1. Architecture details of GPUs and CPU Used

3.2 Micromagnetic Model

3.2.1 Micromagnetics

Micromagnetism [25, 26, 27] is a generic term used to study the fundamental magnetization processes (the interactions between the magnetic moments of ferromagnetic body) on microscopic space and time scale[28]. In general these interactions under different conditions govern the behavior of a magnetic body.

3.2.2 Continuum hypothesis

Let us consider a small volume dVr on a body having magnetic region Ω where r$\epsilon\Omega$ and it represents the position vector in the volume dVr. The region dVr contains N number of magnetic moments μ_j where j= $1, ...N$, while dVr is small enough that the average magnetic moment varies smoothly. The product of Magnetization vector field $\mathbf{M}(\mathbf{r})$ and the elementary volume dVr gives the net magnetic moment of elementary volume dVr, $\mathbf{M}(\mathbf{r})$ gives the average of total magnetic moments in the small volume dVr in a ferromagnetic body. According to continuum hypothesis we can see that

$$\mathbf{M}(\mathbf{r}) = \frac{\sum_j^N \mu_j}{dV_r} \tag{3.1}$$

Moreover, magnetization is also a function of time t,[29].

$$\mathbf{M} = \mathbf{M}(\mathbf{r},t) \tag{3.2}$$

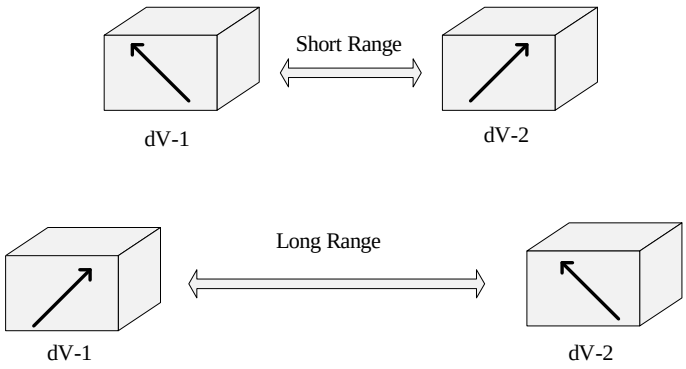

Figure 3.1. Short and long range magnetic interactions

3.2.3 Micromagnetic free energy

The interactions between the magnetic moments in a ferromagnetic body are managed by different competing short and long range energy terms among them the four important contributions to the Landau free energy of a ferromagnetic body are the exchange energy, the magneto crystalline anisotropy energy, the magnetostatic energy, and the Zeeman energy in an external field[30].

According to second law of thermodynamics the change in the Gibs free energy of a ferromagnetic body must hold the following inequality

$$\triangle E \quad = \quad E_{final} - E_{initial} \leq 0 \tag{3.3}$$

Where $E_{initial}$ and E_{final} are the initial and final Gibs free energy in a ferromagnetic body respectively. Equation 3.3 shows that Gibbs free energy in a ferromagnetic body tends to be decrease towards zero and is minimum at equilibrium condition. In a ferromagnetic body the Gibbs free energy is important to determine the behavior of magnetization vector $\mathbf{M}$ for example as discuss above at equilibrium state in a ferromagnetic body the Gibbs free energy of the system is at its minimum.

As shown in figure 3.1 we can categorize the magnetic interaction between the magnetic moments in to two main groups.

- Short range (maxwellian) interactions between magnetic moments.
- Long range (maxwellian) interactions between magnetic moments.

Short range interactions between magnetic moments

Exchange and Anisotropy interactions come in the category of short term (maxwellian) interactions. The energy of the short range interaction depends only on the electron spins of neighboring elements [31]. The compute complexity of all the short range interactions is $O(N)$ therefore their computation is not a big problem with respect to both time and memory consumption.

Long range interactions between magnetic moments

On the other end, long range components are calculated for each element of interest against all points in the discretized magnetic body. These are essentially convolution operations corresponding to a complexity of $O(N^2)$ and is the main contribution towards the total simulation time. By shifting the convolution to be performed in frequency domain, the complexity can be reduced to $O(NlogN)$. The main contribution here is the forward and inverse multidimensional Fourier transforms.

3.2.4 The Dynamic Landau-Lifshitz-Gilbert Equation

Classical iterative methods [32, 33] can be applied to solve numerically the Brown's equations [26] which are used to find the equilibrium configuration of the magnetization within the body. But the issue in solving these equation using classical iterative methods is that they do not reflect the actual evolution of magnetization during time. Hence in order to find that how this equilibrium reaches over time we require new equation which describes the motion of the magnetization over time. The dynamic model was first given by landau and Lifshitz [25] and later on modified by Gilbert [34, 35]. The LLG (Landau-Lishitz-Gilbert) equation is the dynamic model for the precessional motion of the magnetization M which is exposed to an effective field H_{eff} over time.

when a magnetic field $\overrightarrow{H}$ is applied to magnetic material it exerts a on magnetic moment $\overrightarrow{M}$, which is equal to

$$\mathbf{M} = \overrightarrow{M} \times \mu_0 \overrightarrow{H} \tag{3.4}$$

In a non equilibrium condition that is when ($M \neq 0$) the magnetic moment $\overrightarrow{M}$ has a gyroscopic reaction which is described by the equation

$$\frac{\partial \overrightarrow{M}}{\partial t} = -\gamma \mu_0 (\overrightarrow{M} \times \overrightarrow{H}) \tag{3.5}$$

where, γ represents the gyromagnetic factor, its absolute value is $= 2.21 \times 10^5 mA^{-1}s^{-1}$of the ratio

$$\gamma = \frac{g \mid e \mid}{2m_e c} \tag{3.6}$$

where g is the Lande splitting factor whose value is $\backsimeq 2$, e is the electron charge whose value is $e = -1.6 \times 10^{-19} C$,$m_e = 9.1 \times 10^{-31} kg$ is the electron mass and c is the speed of light whose value is $c = 3 \times 10^8 m/s$.

Equation 3.5 describes the precession of the magnetization $\overrightarrow{M}$ around the effective field $\overrightarrow{H}$ as shown in figure 3.2. If the external applied field is sufficiently large, the magnetization tends to align parallel to the field regardless of the initial magnetic state and with the passage of time saturation reaches and the precession stops as $\overrightarrow{H_{eff}}$and $\overrightarrow{M}$ becomes parallel and

$$\overrightarrow{M} \times \overrightarrow{H} = 0 \tag{3.7}$$

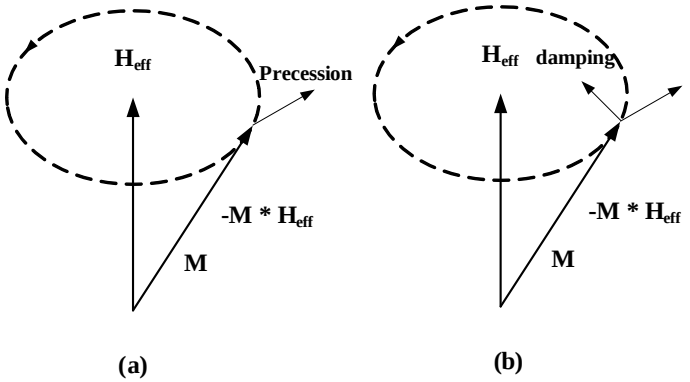

Figure 3.2. (a)Undamped gyromagnetic precession, (b) Damped gyromagnetic precession

From equation 3.5 we cannot deduce the change in the slope of magnetization relative to the field. Hence a Rayleigh dissipative term is introduced in a phenomenological way.

$$\frac{\partial \overrightarrow{M}}{\partial t} = -\gamma(\overrightarrow{M} \times \mu_0 \overrightarrow{H}) + \frac{\alpha}{M_s}(\overrightarrow{M} \times \frac{\partial \overrightarrow{M}}{\partial t}) \tag{3.8}$$

Equation 3.8 is known as the Gilbert equation, where α is the damping coefficient which shows the rate of energy loss, It comprising all the energy loss and its value depends on the material. By applying the limit of low damping on Gilbert equation 3.8 as proposed by Landau and Lifshitz [33] we get Landau and Lifshitz in Gilbert form

$$(1+\alpha^2)\frac{\partial \overrightarrow{M}}{\partial t} = -\gamma(\overrightarrow{M} \times \mu_0 \overrightarrow{H}) + \frac{\alpha\gamma}{M_s}[\overrightarrow{M} \times (\overrightarrow{M} \times \mu_0 \overrightarrow{H})] \tag{3.9}$$

Equation 3.9 is valid for external field $\overrightarrow{H}$, and can be generalized in case of the local field H_{eff} in equation 3.12, so the equation 3.9 reduced to

$$\frac{\partial \overrightarrow{m}}{\partial t} = -(\overrightarrow{m} \times \overrightarrow{H_{eff}}) - \alpha[\overrightarrow{m} \times (\overrightarrow{m} \times \overrightarrow{H_{eff}})] \tag{3.10}$$

Similarly (LLG) equation 3.8 would also be reduced to equation 3.11 which is normalized form of LLG equation.

$$\frac{\partial \vec{m}}{\partial t} = -\vec{m} \times \overrightarrow{H_{eff}} + \alpha \vec{m} \times \frac{\partial \vec{m}}{\partial t} \tag{3.11}$$

The damping coefficient α, describes the overall decrease in the total energy of the ferromagnetic system through various relaxation mechanisms. In case of ferromagnetic materials α has no constant value and may depend on non-linear magnetization. Usually, in calculations, simplifying assumptions fix the value of α between 0.1 and 1. A value of α close to critical damping contributes to the increase in computing speed. While when the problem is reduced to micromagnetic equilibrium states the term drift is neglected and we find the approximation of infinite damping. This approximation has been implemented as an algorithm for energy minimization [33, 36].

3.2.5 Effective Magnetic Fields

The change of the magnetization is due to the effective magnetic field[37] H_{eff} which can have several contribution as shown in equation 3.12 .

$$\mathbf{H_{eff}} = \mathbf{H_{exc}} + \mathbf{H_{anis}} + \mathbf{H_{ext}} + \mathbf{H_m} \tag{3.12}$$

In equation 3.12 all short range components can be calculated using interactions between direct neighbors to a point of interest, resulting in complexity of $O(N)$. On the other end, long range components are calculated for each element of interest against all points in the grid. These are essentially convolution operations corresponding to a complexity of $O(N^2)$ and is the main contribution towards the total simulation time.

In micromagnetic simulation, the magnetostatic field $\mathbf{H_m}$ is typically a long-range interaction because its computation at a given point involves contribution of the whole magnetization vector field and it holds the following equation.

$$\mathbf{H}_m(r) = \frac{1}{4\pi}\int_V \frac{(r-r')}{|r-r'|^3}.(-\nabla.\mathbf{M}(r'))dV_{r'} + \frac{1}{4\pi}\int_{\partial V} \frac{(r-r')}{|r-r'|^3}.[\mathbf{M}(r').n'(r')]dS_{r'} \tag{3.13}$$

Where the integration domain V corresponds to the ferromagnetic body and ∂V is the body surface. After spatial discretization, equation 3.13 can be numerically computed and the field at each element/cell is evaluated as a function of the N point/cell magnetization. This results in $O(N^2)$ operations.

Equation 3.13 involves a convolution operations and by shifting the convolution to be performed in frequency domain, the complexity can be reduced to $O(NlogN)$. The main contribution here is the forward and inverse multidimensional Fourier transforms.

In micromagnetic simulations, very fine time and space dicretization, makes it very large numerical problem, but fortunately micromagnetic codes in general are suitable for parallel programming as it can be easily divided in to independent parts which can run in parallel[21]. GPUs provide the best solution for such problems with respect to price performance ratio.

The study and observation of magnetization behavior at sub-nanosecond timescales is crucial to a number of areas such as magnetic sensors, non volatile storage devices and magnetic nanowires etc. Since micromagnetic codes in general are suitable for parallel programming as it can be easily divided into independent parts which can run in parallel, therefore current trend for micromagnetic code concerns shifting the computationally intensive parts like the magnetostatic field calculation as discussed above to GPUs.

All the current micromagnetic solvers on GPU are CUDA based and uses the general-purpose FFT library (cufft) for the computation of magnetostatic field. This limits the current GPU based magnetostatic solver

1. To NVIDIA based hardware only

2. By the use of general-purpose FFT library they can not fully exploit the zero padded input data with out transposition and symmetries inherent in the field calculation

3. On single GPU the input problem size is also an issue.

My PhD work mainly focuses on the development of highly parallel magnetostatic field solver for micromagnetic simulators on GPUs. I am using OpenCL for GPU implementation, with consideration that it is an open standard for parallel programming of heterogeneous systems for cross platform. It targets different devices such as GPUs by different vendors such as Nvidia, ATI and Intel etc, along with CPU and other processing hardware which conform to its specification. The magnetostatic field calculation is dominated by the multidimensional FFTs (Fast Fourier Transform) computation. Therefore I have developed the specialized OpenCL based 3D-FFT library for magnetostatic field calculation which made it possible to fully exploit the zero padded input data with out transposition and symmetries inherent in the field calculation. As a result the complexity of overall system reduced significantly compared to current GPU based solvers. Moreover it also provides a common interface for different vendors' GPUs. In order to fully utilize the GPUs parallel architecture my solver handles many hardware specific technicalities such as coalesced memory access, data transfer overhead between GPU and CPU, GPU global memory utilization, arithmetic computation, batch execution etc.

In the second step to further increase the level of parallelism and performance, I have developed a parallel magnetostatic field solver on multiple GPUs. Utilizing multiple GPUs avoids dealing with many of the limitations of GPUs (e.g., on-chip memory resources) by exploiting the combined resources of multiple on board GPUs.

3.3 Magnetostatic Field Computation

The concept that how the magnetic moments in a magnetic body interacts over a long distance comes from the magnetostatic interactions. The magnetic field at a given point P_1 does not depends only on magnetization vector field at that point, rather it depends on a all the magnetization vector field distribution in a magnetic body. The property of magnetic material to lift the object against the force of gravity is due to magnetostatic energy [38]. The concept of magnetostatic field H_m can be explained with the help of Maxwellian's equations for magnetized media.

$$\nabla.H_m = -\nabla.M \; inside Volume$$

$$\nabla.H_m = 0 \; outside\, Volume \tag{3.14}$$

$$\nabla \times H = 0$$

and at the body discontinuity surface i.e.

$$n.[H_m]_{\partial V} = n.M \tag{3.15}$$

$$n.[H_m]_{\partial V} = 0$$

In equation 3.14 and equation 3.15 n is the outward normal to the boundary ∂V of the magnetic body, and $[H_m]_{\partial V}$ is the jump of the vector field $H_m$over the ∂V.

Maxwellian's equations for magnetized media 3.14 and 3.15 shows the relationship between magnetostatic field and the magnetization in a ferromagnetic body which we can summarize as follow.

From these equations one can deduce that the magnetostatic field at any given point "r" in a ferromagnetic body depends on the magnetization "M" of all the points in a discretized ferromagnetic body. Thus the magnetostatic field is a consequence of long range maxwellian interaction in a magnetic body. From the point of view of computational complexity, let us consider a magnetic body discretized into N cells then the magnetostatic field computation for N cells would require N^2 operations. While all other terms in effective magnetic field as discussed above requires N operations. Therefore in micromagnetic simulations the computation of magnetostatic field due to its complexity is the most time consuming part of the simulation. Secondly with respect to memory consumption, magnetostatic field computation is also a huge memory demanding problem.

Secondly maxwellian equation 3.14 and 3.15 suggest that magnetostatic field

computation is an open boundary problem. This means that for calculating the magnetostatic field at a given point depends on the all the points in the whole space, which is obviously not possible. Therefore quoting Aquino [29] .... "Numerical methods consistent with the continuum model have to be used in numerical simulations", like the FFT discrete convolution method which I am using for the magnetostatic field computation on GPUs.

3.4 FFT discrete convolution method

When we talk about solving the partial differential equations we come across two well know methods that are

1. Finite Differences Method

2. Finite Element Method

The main difference between these two method is that finite difference method is mainly used for regular shapes or structured meshes where the observing sample can be discretized into equal cuboid cells, but by using different techniques it can also be used for irregular shapes as well. While on the other hand finite element method can be used for both structured and un-structured meshes.

FFT discrete convolution method is used in the case of regular meshes approach, based on finite difference method. Regarding the magnetization in each discretized cell, in literature there exist two approaches. In the first approach the magnetization within each cell is considered to be constant, this approach is referred as constant volume charges. While in the second approach the magnetization M within each discretized cell of ferromagnetic body is assumed to be uniform which means.

$$\nabla.\mathbf{M}(r') = 0 \tag{3.16}$$

McMichael in [39] gives the comparison of these two approaches.

Now in order to explain the FFT discrete convolution method lets start from the integral form of magnetostatic field as discussed by H. Neil Bertram in [40] that is

$$\mathbf{H}_m(r) = \frac{1}{4\pi}\int_V \frac{(r-r')}{|r-r'|^3}.(-\nabla.\mathbf{M}(r'))dV_{r'} + \frac{1}{4\pi}\int_{\partial V} \frac{(r-r')}{|r-r'|^3}.[\mathbf{M}(r').n'(r')]dS_{r'} \tag{3.17}$$

In equation 3.17 r' is the source point in a discretized magnetic body while the r is the observation point. The magnetostatic field $\mathbf{H}_m(r)$ is directed from source to observation point in a discretized magnetic body with magnitude $(-\nabla.\mathbf{M}(r'))dV_{r'}$for the volume and $[\mathbf{M}(r').n'(r')]dS_{r'}$for the surface of magnetic body divided by the square of distance.

Where in the equation 3.17 the first part calculates the integral over the volume V of magnetic body while the second part calculates the integral over the surface ∂V. The n' is the outward normal to the surface ∂V of the magnetic body at the source point r' hence the magnetic surface charge density is

$$\mathbf{M}(r').n'(r')] \tag{3.18}$$

As discussed above there exist two approaches for magnetostatic field calculation. In the first approach the magnetization within each cell is considered to be constant, this approach is referred as constant volume charges. While in the second approach the magnetization M within each discretized cell of ferromagnetic body is assumed to be uniform which means $\nabla.\mathbf{M}(r') = 0$, therefore the magnetostatic field at each point of discretized magnetic body can be evaluated due to surface charge densities of all other cells in a discretized magnetic body.

For a discretized magnetic body the equation 3.17 remains unchanged [29]. As shown in figure 3.3 for a discretized ferromagnetic body sample let us consider that total number of discretized cells in a ferromagnetic body are N with with number of cell $n_x$along X direction, $n_y$along Y direction and $n_z$along Z direction in the Cartesian coordinates (i.e.$N = n_x \times n_y \times n_z$). The combination of three indexes $i.j, k$ uniquely represents each cell in a discretized magnetic body where

$$i = (0,1,2.....n_x - 1) \ \ and \ j = (0,1,2.....n_y - 1) \ and \ k = (0,1,2.....n_z - 1) \quad (3.19)$$

now with these assumptions the expression for a magnetostatic field $\mathbf{H}_m$ for a cell (i, j, k) in a discretized magnetic body can be represented as a discrete convolution like

$$\mathbf{H}_m(i,j,k) = \sum_{i'=0}^{n_x-1} \sum_{j'=0}^{n_y-1} \sum_{k'=0}^{n_z-1} = \mathbf{N}_{(i-i',j-j',k-k')} \,.\, \mathbf{M}_{(i',j',k')} \quad (3.20)$$

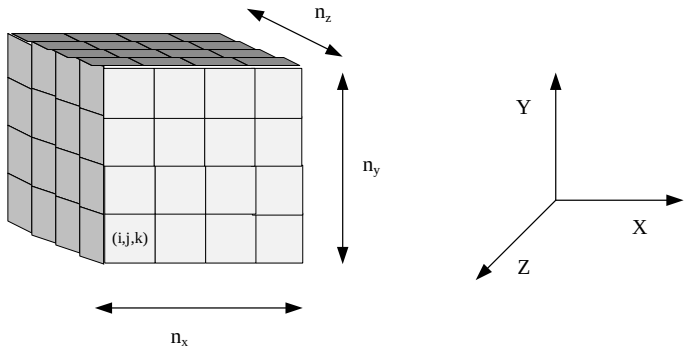

Figure 3.3. Ferromagnetic body discretization along$X, Y and Z$ direction with number of cell $n_x$along X direction, $n_y$along Y direction and $n_z$along Z direction in the Cartesian coordinates

In equation 3.3 $\mathbf{N}_{(i-i',j-j',k-k')}$ is a demagnetization tensor. It depends on the shape or geometry of magnetic body and the relative position of two cells $n_{(i,j,k)}$ and $n_{(i',j',k')}$in a whole discretized magnetic body as shown in figure 3.4. In literature different methods have been proposed for the calculation of demagnetization tensor [41, 42, 43] but the most accurate and widely used method is given by Newell [42]. In Cartesian coordinates (X, Y, Z) the demagnetization tensor is represented by 3×3 matrix for three dimensional discretized magnetic body.

$$\mathbf{N}_{(i-i',j-j',k-k')} = \begin{pmatrix} N_{xx'} & N_{xy'} & N_{xz'} \\ N_{yx'} & N_{yy'} & N_{yz'} \\ N_{zx'} & N_{zy'} & N_{zz'} \end{pmatrix} \tag{3.21}$$

In expression 3.21 each component in the demagnetization tensor matrix is calculated by the interaction between two pairs of rectangular surfaces one from each source point and the observation point, which are perpendicular to the considered direction of the component in Cartesian coordinates [42]. For example if we want to calculate the $N_{xx'}$ component, the surfaces which are perpendicular to x and x' in both source point and the observation point are the yz surfaces. Therefore the computation of $N_{xx'}$component would only involve the interaction between the yz surfaces of the source point and the observation point. Similarly the computation of $N_{xy'}$component would involve the yz surfaces in the observation point and xz surfaces in the source point.

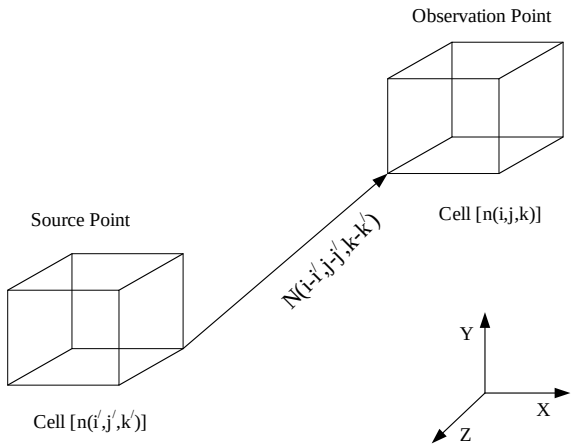

Figure 3.4. Demagnetization Tensor $N_{(i-i',j-j',k-k')}$

3.4.1 Compute Complexity

Based on equation 3.20 if we want to compute the magnetostatic field for each point of a discretized magnetic body it would require a compute complexity of $O(N^2)$. The $O(N^2)$ is required because each element in magnetostatic computation requires a contribution from N (where $N = n_x \times n_y \times n_z$ as shown in figure 3.3) cells which

is the total number of cells in a discretized magnetic body. The performance of micromagnetic solver heavily depends on the method used to calculate the demagnetization field $\mathbf{H_m}$ given in equation 3.20. It is a discrete convolution problem and can be solved using discrete fourier transform by implementing the efficient and well established Fast Fourier Transform (FFT) algorithm.

Therefore in my implementation I used Fast Fourier Transform based method to solve the discrete convolution equation. By shifting the convolution to be performed in frequency domain, the complexity can be reduced from $O(N^2)$ to $O(N \log N)$ where N is the number of simulation cells.

The expanded form of discrete convolution theorem in equation 3.20 is

$$\begin{aligned}
\tilde{\mathbf{H}}_{x(i,j,k)} &= \tilde{\mathbf{N}}_{xx'(i,j,k)}\tilde{\mathbf{M}}_{x(i,j,k)} + \tilde{\mathbf{N}}_{xy'(i,j,k)}\tilde{\mathbf{M}}_{y(i,j,k)} + \tilde{\mathbf{N}}_{xz'(i,j,k)}\tilde{\mathbf{M}}_{z(i,j,k)} \\
\tilde{\mathbf{H}}_{y(i,j,k)} &= \tilde{\mathbf{N}}_{yx'(i,j,k)}\tilde{\mathbf{M}}_{x(i,j,k)} + \tilde{\mathbf{N}}_{yy'(i,j,k)}\tilde{\mathbf{M}}_{y(i,j,k)} + \tilde{\mathbf{N}}_{yz'(i,j,k)}\tilde{\mathbf{M}}_{z(i,j,k)} \\
\tilde{\mathbf{H}}_{z(i,j,k)} &= \tilde{\mathbf{N}}_{zx'(i,j,k)}\tilde{\mathbf{M}}_{x(i,j,k)} + \tilde{\mathbf{N}}_{zy'(i,j,k)}\tilde{\mathbf{M}}_{y(i,j,k)} + \tilde{\mathbf{N}}_{zz'(i,j,k)}\tilde{\mathbf{M}}_{z(i,j,k)} \qquad (3.22)
\end{aligned}$$

In above equation, the FFT quantities are with tilde sign. The steps to calculate the demagnetizing field $H_m$can be summarized as follows:

- First of all, the FFTs of six instead of nine (due to symmetries e.g $N_{x,y} = N_{y,x}$) demagnetizing tensors instead of nine given in equation 3.21 have to be performed and stored in the memory.

- For each computation of the demagnetizing field, six FFTs has to be computed, three related to magnetization vectors, $(Mvec)$ namely $\mathbf{M_x}$, $\mathbf{M_y}$ and $\mathbf{M_z}$ and three inverse FFTs of the $(Hvec)$ components $\mathbf{H_x}$, $\mathbf{H_y}$ and $\mathbf{H_z}$ and these have to be computed for each time step.

3.5 Current state of the art of micromagnetic solvers

In this section I would discuss the current state of the art of micromagnetic solvers mainly focusing on GPUs based solvers. In micromagnetic simulations, the magnetostatic field calculation is the most time and memory consuming part of the simulation [22, 23] and is computed many times at each time step interval. Hence

even with the fast computing methods for magnetostatic field calculation, such as FFT are limited in their performance when implemented on single CPU or even on multiple CPUs only for very large input problem sizes. Hence the parallel techniques are required in order to perform the large micromagnetic simulations. Therefore all of the current micromagnetic solvers in one or other way uses different parallel architectures discussed in section 2.2 to design efficient parallel algorithms as shown in Table 3.2 [44].

3.5.1 GPU based solvers

Recently, different groups developed micromagnetic simulators on graphics hardware and shown a substantial speedup compared to CPU based simulators such as [45, 46, 47, 48].

Mu-Max [45] is CUDA based general purpose micromagnetic solver and it works only for single precision floating point. It uses CUDA based cufft library for the computation of magnetostatic field. It shows a speedup of over a factor 100x compared to CPU-based OOMMF running on a single core CPU. In the case of Mu-Max the high speedup is achieved at the cost of small micromagnetic input problem sizes. The global memory on GPU is a scarce resource. Small problem sizes which can fit on a whole in to global memory of GPU mitigates the expensive intermediate

Table 3.2. Classification of current Micromagnetic Solvers based on numerical methods, Architectures (Shared Memory/ Distributed Memory/ GPU) and Language/ API

Name	Numerical Method	Architecture	Language/API
Mu-Max [45]	Finite Difference	GPU	CUDA/cufft
FastMag [46]	Finite Element	GPU	CUDA/cufft
GPMagnet[47]	Finite Difference	GPU	CUDA/cufft
TetraMag[[48]	Finite Element	GPU	CUDA
OOMMF[23]	Finite Difference	SM	C++/TCL
Nmag [49]	Finite Element	DM	Python/MPI
Magpar [50]	Finite Element	DM	C++/MPI
M^3[51]	Finite Difference	SM	Matlab

data transfer cost between GPU and CPU. Secondly it also reduces the overhead of multiple invocation of the kernel side code on GPU. As a result shows a tremendous performance by compromising on input problem size. On the other hand in most of the cases like in micromagnetic simulation, when the problem size becomes so large not to accommodate as a whole on GPU memory, we have to perform expensive (with respect to time) data transfer between CPU and GPU and kernel calls. While my implementation can handle very large problem sizes by dividing the whole problem in to independent parts so that at a time the whole global memory of GPU is available only to that part. I am using different optimization strategies as discussed in section 3.7 to mitigate the overhead of CPU to GPU data transfer time.

The FastMag [46] (Fast Micromagnetic simulator) is finite element based general purpose micromagnetic solver developed at center for Magnetic Recording research and Department of electrical and computer engineering, university of California, San Diego. It uses nonuniform grid interpolation method (NGIM) to compute the magnetostatic field with complexity $O(N)$. It shows a GPU to CPU speedup of two order of magnitude.

GPMagnet [47] is also general purpose finite difference based GPU micromagnetic solver. It also uses CUDA based cufft library and have shown a speedup of two orders of magnitude with respect to its equivalent serial implementation.

TetraMag [48] like FastMag is finite element micromagnetic simulation tool for GPUs. It works with double precision floating point accuracy and is based on CUDA architecture. It demonstrates a speedup factor of up to four on single GPU compared to equivalent CPU implementation using eight cores. All these above mentioned GPU based micromagnetic solvers are heavily dependent on the CUDA based cufft library which limits the usage of these codes to NVIDIA based hardware only.

3.5.2 Limitations of current solvers

On the other hand I have developed OpenCL based magnetostatic field solver, where OpenCL works across heterogeneous platforms consisting of CPUs, GPUs, and other processors which conform to its specification[18, 19]. Where CUDA specifically targets GPU devices only. Secondly I have developed my own OpenCL based 3-D

FFT library which gives the freedom to deeply manipulate the transform according to the requirements to handle the specific properties of input data, which cannot be done with general purpose FFT library such as cufft library. At the time of writing and in best of my knowledge it is the first OpenCL based magnetostatic field solver.

3.6 OpenCL based GPU implementation

3.6.1 Symmetry Properties of Components

Since the FFT is well suited for periodic input, the initial data structure is modified to include zero-padding to avoid the effect of circular convolution. In fact zero padding (Figure3.5) is also necessary to get the correct out put in real space after performing the inverse FFT. With zero padding the input size to micromagnetic solver increases eight times, as if you have a initial grid dimensions as $N_x \times N_y \times N_z$, after zero padding it would become $2N_x \times 2N_y \times 2N_z$, therefore at the end the each FFT size would be double the initial size and the total number of FFTs would be eight times the initial value.

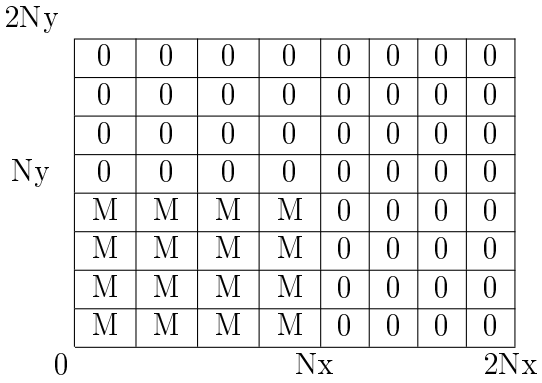

Figure 3.5. 2-D magnetization Vector Zero Padding

There are present a few symmetries inside both the demagnetizing tensor and the magnetization vector. Since the input data is real, conjugate symmetries are present in the output. In addition, the demagnetizing tensors are either odd/even symmetric. Details of these are discussed in each section. These are listed in Table-3.3.

Input $x\,(n)$	**Output Properties** $X\,(n)$
real	$X^*\,(-n)$
real & even	$X\,(-n)$ pure real
real & odd	$X^*\,(-n)$
imag & even	$X\,(-n)$ pure imag
imag & odd	$-X\,(-n)$

Table 3.3. Symmetry Properties of Fourier Transform

3.6.1.1 Demagnetizing Tensor

The tensor components outlined in Eq-3.21 consist of either odd or even symmetries. The diagonal consisting of the xx, yy, and zz contain pure even symmetries, whereas the non-diagonals contain mixed symmetries. For the diagonals, e.g., the xx component, based on the list of symmetry properties listed in Table-3.3, we obtain the following after each transform:

$$\begin{aligned} X\,(I,j,k) &= X\,(-I,j,k) && Real + Even \\ Y\,(I,J,k) &= Y\,(-I,-J,k) && Real + Even \\ Z\,(I,J,K) &= Z\,(-I,-J,-K) && Real + Even \end{aligned}$$

For the non-diagonals, e.g., the xy component, containing odd symmetries along x and y but even in z, we obtain the following:

$$\begin{aligned} X\,(I,j,k) &= X^*\,(-I,j,k) && Imag + Odd \\ Y\,(I,J,k) &= -Y\,(-I,-J,k) && Real + Odd \\ Z\,(I,J,K) &= Z^*\,(-I,-J,-K) && Real + Even \end{aligned}$$

Following the same approach for other components gives the result that the transform of the demag tensor is purely real, and contains the same odd/even symmetries as that of its input.

3.6.1.2 Magnetization and magnetostatic field

The magnetization vector contains the zero-padded magnetization vector (Fig-3.5). Since the initial data is real, we obtain the following symmetries in the output:

$$\begin{aligned} X(I,j,k) &= X^*(-I,j,k) \quad in\ Strip \\ Y(I,J,k) &= Y^*(-I,-J,k) \quad in\ Plane \\ Z(I,J,K) &= Z^*(-I,-J,-K) \quad in\ Volume \end{aligned}$$

The symmetries are translated to the magnetostatic field as well. Section 3.7describes in detail the complexities and the savings in each axis transform based on zero padded input data and the symmetries inside the demagnetizing tensor, magnetostatic field and the magnetization vector.

3.7 Implementation Approaches

3.7.1 GPU-Optimized Implementation

Over the past few years, graphic cards have started to be used for general purpose computing in addition to computer graphics. The massive parallelism offered by these cards are exploited by applications involving large number of calculations. Scientific applications are amongst the greatest beneficiaries.

The GPU itself is a many-core processor where dozens of streaming processors with hundreds of cores support thousands of threads [52], all of which run concurrently running concurrently on single chip. The core hierarchy is depicted in Figure-3.6, showing a medium-range NVIDIA based graphics card. Thread management at such hardware level requires context-switching time close to null otherwise penalizing performance. Since the GPU hardware can be classified as SIMT (single-instruction, multiple threads), therefore general purpose CPU-bound applications which have significant data in-dependency are well suited for such devices. Performance evaluation with respect to GFLOPS shows that GPUs outclass its CPU counterparts by manifolds. A high-end Core-I7 Desktop processor (3.46 GHz) can deliver a peak of 55.36 GFlops as compared to Nvidia Quadro 6000 which gives peak performance of 1030 GFlops. Table-1 reports some architecture details of GPUs and Intel Core2 system that we have used for our implementation of magnetostatic field solver. The devices include a high-end graphics card like Quadro 6000 comprising of 14 stream processors with 32 cores each, and NVIDIA GTX 260 with 27 processors having 8 cores each [24].

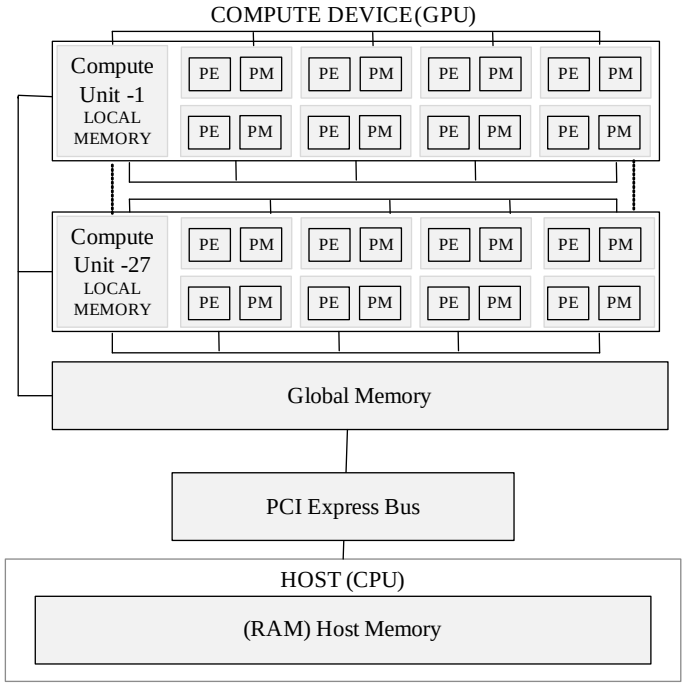

Figure 3.6. NVIDIA GTX-260 GPU Architecture

3.7.1.1 Minimizing Data Transfer Between CPU and GPU

One of the major concerns for the performance of an application on GPU is the data transfer between host (CPU) and device (GPU). Most of the recent GPUs have their own dedicated Global memory with high bandwidth like for example Nvidia GTX 280 have 141 GBps global memory bandwidth while on the other hand the communication between the GPU and CPU takes place on PCI express bus which has normally a bandwidth typically a few GBps a sketch of general GPU memory hierarchy is depicted in figure3.6. In our magnetostatic field solver on GPU we specially considered this constraint. We transfer all the data of each component of magnetization matrices and the inverse FFTs given in equation 3.22 at the beginning of simulation to global memory of GPU and perform 3-D FFT while keeping the intermediate results of each 1-D transform on GPU global memory and from CPU side we only generates the instruction to control the execution sequence, and finally copy backs the results at the end of each 3-D FFT.

As discussed in 3.4.1 For each computation of the demagnetizing field, six FFTs

has to be computed, three related to magnetization matrices, namely M_x, M_y and M_z and three inverse FFTs of the components $H_{m,x}$, $H_{m,y}$ and $H_{m,z}$ and these have to be computed for each time step. Therefore we would mainly focus on the optimization of these components. While the six demagnetization tensor values given in equation 3.21 have to be calculated only once in the simulation and stored in the memory.

In order to avoid the end effect of circular convolution the initial data structure is modified to include zero padding, as a result the input size of a problem to micromagnetic solver increase by a factor of eight, for simplicity zero padding of 2-D grid is shown in figure3.5. It is completely useless to transfer the zero padded data. In our implementation we do this zero padding at GPU side before performing the FFTs on magnetization vector. Secondly the input data in case of magnetization vector is real. Therefore the input imaginary values are always zero. For these imaginary values we just assign a memory location on GPU side and initialize it with zero values. As a result we reduces the input data transfer to GPU in case of magnetization vector by half. On the other hand in the inverse FFTs of H_m Vectors the data transfer is reduced by half by considering the complex conjugate symmetries present in the data. Our results show the significance of this data transfer overhead in the overall performance of the simulation.

3.7.1.2 Coalesced Memory Access

The coalesced memory access is the most important consideration for performance while programming on GPUs. The NVIDIA Quadro 6000, built on innovative NVIDIA fermi architecture, supports 14 microprocessors having 32 cores each, thus resulting into 448 cores in total, arranged as array of streaming multi-processors. This means that on Quadro 6000 GPU can run 448 concurrent threads. In order to exploit this huge parallelism that can be achieved on GPU architecture, the global memory on GPU must be accessed in a coalesced way. On GPU each thread does not access the global memory individually rather group of treads called half wrap (16 threads) access the global memory simultaneously as shown in figure 3.7 (a), resulting in a single memory transaction under certain access requirements. On the other hand as shown in figure 3.7 (b) when there is a stride (stride of one in this

case) in the data then the data required by the half of the active threads is not in the fetched 64B aligned data segment therefore second memory transaction would be performed. In worst case when the stride is greater than the half warp size, 16 different memory transactions would be required resulting a overall performance degradation. Rest of this section explains that how we achieved the coalesced global memory access in our implementation.

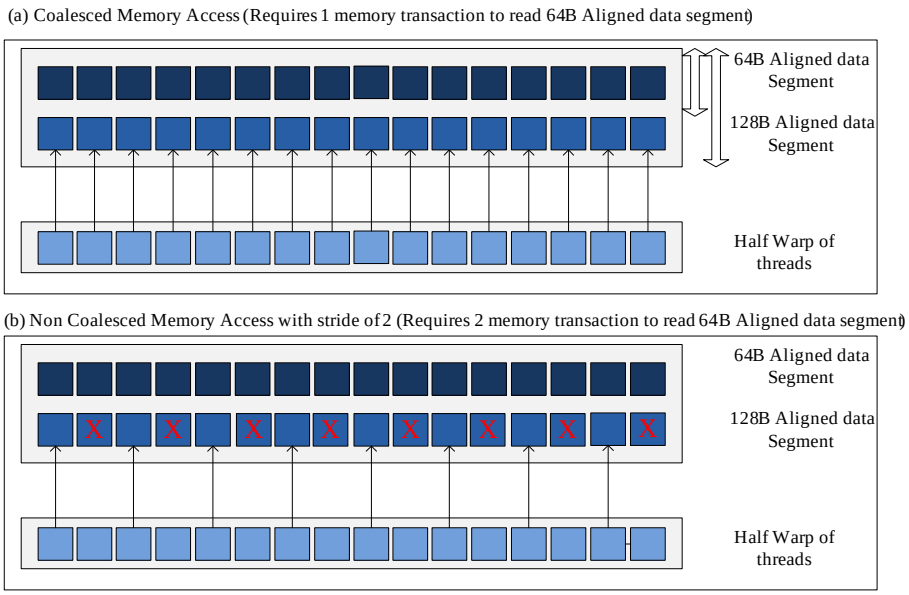

Figure 3.7. Complex Number Coalesced Memory Access

Memory Coalescing In 3-D FFT Transforms: Figure 3.8 depicts the arrangement of input data to 3-D FFT in global memory of GPU. As we can see from the figure 3.8 while performing the transform along x-axis the stride in accessing the data from global memory is zero therefore no need to change the data arrangement. On the other hand while performing the transform along y-axis the stride in the data in equal to the x-range of 3-D input data. In this case when the x-range becomes greater than the size of half warp, 16 different memory transactions would be performed to access the required data to perform single transform along y-axis. While in case of transform along z-axis the situation is even more worst where the stride is equal to xy-plane. There can be different solutions to avoid the strided memory access. In general 3-D FFT uses transposition to perform set of 1-D FFT separately on x, y and z direction by rearranging the data along x-axis. In this case transposition is extra

overhead on the performance, but the data becomes contiguous in the global memory for coalesced access. In the current implementations of micromagnetic solvers where they are using cufft library to achieve the coalesced memory access, the transposition of the data becomes unavoidable. In our implementation by developing our own 3-D FFT library for GPU we are able to avoid the overhead of transposition and also accessing the global memory in a coalesced way, resulting a very high performance. Consider the transform along y-axis while keeping the current arrangement of the data. For a simplicity lets take an example of 2-D grid as shown in figure 3.5. Instead of non coalesced fetching of data along y-axis, we move along x-axis to access the data in coalesced way and perform the computation of all FFTs along y-axis on the data that is along x-axis. The same procedure in adopted for z-axis transform.

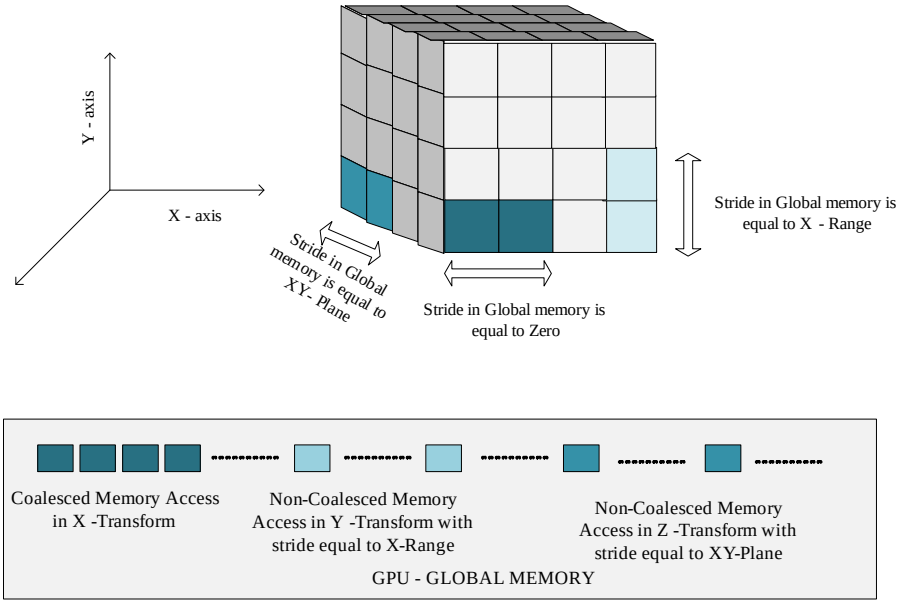

Figure 3.8. Memory Coalescing in Transforms

3.7.1.3 Minimizing GPU memory Utilization

Memory on GPUs is normally a scarce resource typically for micromagnetic simulations where the inputs sizes are normally very large to fit into the GPU memory. In our implementation we perform in-place bit reversal as well as the FFTs which drastically reduce the memory consumption. Secondly we also save the memory

by avoiding the transposition which is necessary in case of using CUDA 1-D FFT library. As the result of micromagnetic simulation is real hence in an inverse transform we only save the real values, which do not only save the memory but also the data transfer overhead from device to host.

3.7.1.4 Minimizing the Arithmetic Computation

In CUDA and OpenCL two types of math operations are supported. Functions using native as a prefix map directly to the hardware level. Native functions are faster but are less accurate. While the regular functions are slower but have higher accuracy. The throughput of native sin and cos functions is 1 operation per clock cycle, while regular sin and cos functions are much more expensive and become even more expensive when the absolute input value to these functions is very small[53], which is common in micromagnetic simulations. On CUDA architecture only single precision floating point native functions are supported hence in my case of double precision we cannot use these native functions, and even in single precision implementation we cannot compromise on accuracy. But we can reduce the use of these slow functions as much as possible. In my implementation I calculate the sin and cos values for only 1 FFT call in x, y and z directions only once at the beginning of the simulation, and then pass these single time calculated values to the kernel performing FFTs on whole data in each time step, resulting a very high performance.

3.7.1.5 Batch Execution

By considering the data dependency problem, one can execute multiple FFTs in Parallel, infect we executes all the FFTs in any dimensions in a single GPU call. By doing so one can save the overhead of invoking kernel for individual FFT call which can have drastic effect with respect to total simulation time for very large 3-D problem sizes where the number of FFTs are large in number. For example let us consider a 3-D problem size of 32^3 if our program does not supports the batch execution it will require 32^3 times the kernel invocation.

3.7.2 FFTs Based Optimizations

For a given volume, the FFT's are performed in batches of 1D FFT's along X, Y, and Z axes in sequence. Based on the symmetry properties outlined in Table-3.3, and due to zero-padded input, reductions in these batches are possible in both computation of the demagnetizing tensor, as well as the magnetization vector field. For computing the demagnetization field at each time step the six FFTs has to be computed, three related to magnetization vectors, ($Mvec$) namely M_x, M_y and M_z and three inverse FFTs for demagnetization field ($Hvec$) components $H_{m,x}$, $H_{m,y}$ and $H_{m,z}$. Any saving in the computation of these FFTs would have an huge effect on the performance gain in overall micromagnetic simulation. The general idea is to minimize the number of batches, and when required, copy the missing information from the present symmetries. In the general case of FFT libraries, which may deal with any kind of input data, the reductions are not in so much detail.

3.7.2.1 OpenCL Based 3-D FFT library on GPUs

3.7.2.2 Savings in Forward 3-D FFT of Magnetization Vectors

As discussed above for three dimensional FFT's zero padding increases the input problem size by a factor of eight (i.e $2N_x \times 2N_y \times 2N_z$), where N_x, N_y, $N_z$are the original dimensional lengths of input array. Since the complexity of FFT is $O(NlogN)$, therefore overall complexity without any optimization on this zero padded data would be $O(8N_xN_yN_z\ log\ 8N_xN_yN_z)$. Let the new dimensional sizes after zero padding are d_x, d_y, d_z then the overall complexity would be $O(d_xd_yd_z\ log\ d_xd_yd_z)$.

The FFT of entirely zero padded sequence is zero, hence there is no need to perform transform on zero padded sequence. Secondly because of the real input there also exist some symmetries as discussed in section 3.6.1 in the input data which can be exploited to reduce the overall number of transforms. For $X - Axis$ transform due to zero padding we would require $\frac{d_y}{2}$ FFT's along $Y - Axis$ instead of d_y and $\frac{d_z}{2}$ FFT's along $Z - Axis$ instead of d_z of size d_x. The complexity of transform along $X - Axis$ would be reduced to $\frac{d_y}{2}\frac{d_z}{2}(d_x\ log\ d_x)$. Hence this would reduces the cost of $X - Axis$ transforms by 75% when $d_x = d_y = d_z$.

As the input data to $X - Axis$ transforms is real hence the output would be conjugate symmetric. Therefore for $Y - Axis$ transforms in order to reduce the

overall complexity we would exploit both the zero padded input data along $Z-Axis$ and the conjugate symmetric data along $X-Axis$. Therefore for $Y-Axis$ transform due to zero padding we would require $\frac{d_z}{2}$ FFT's along $Z-Axis$ instead of d_z and $\frac{d_x}{2}+1$ FFT's due to conjugate symmetry along $X-Axis$ instead of d_z of size d_y. The complexity of transform along $Y-Axis$ would be reduced to $(\frac{d_x}{2}+1)\,(\frac{d_z}{2})\,(d_y\,log\,d_y)$. Hence this would reduces the cost of $Y-Axis$ transforms by almost 75% when $d_x = d_y = d_z$.

Now for $Z-Axis$ transforms the reductions are on the basis of symmetry properties for magnetization vector discussed in section 3.6.1. For $Z-Axis$ transform we would require $\frac{d_x}{2}+1$ FFT's along $X-Axis$ instead of d_x and d_y FFT's along $Y-Axis$ of size d_z. The complexity of transform along $Z-Axis$ would be reduced to $(\frac{d_x}{2}+1)\,d_y\,(d_z\,log\,d_z)$. Hence this would reduces the cost of $Z-Axis$ transforms by almost50% when $d_x = d_y = d_z$ and it would be even more when $d_x > d_y > d_z$.

Finally the total reduction in three dimensional transforms for forward magnetization vector (Figure-3.11) would be given as:

$$\left[\frac{d_y}{2}\frac{d_z}{2}\right]_x + \left[\left(\frac{d_x}{2}+1\right)\frac{d_z}{2}\right]_y + \left[\left(\frac{d_x}{2}+1\right)d_y\right]_z \tag{3.23}$$

and the overall reduction in the complexity would be

$$\frac{d_y}{2}\frac{d_z}{2}\,(d_x\,log\,d_x) \;+\; (\frac{d_x}{2}+1)\,(\frac{d_z}{2})\,(d_y\,log\,d_y) \;+\; (\frac{d_x}{2}+1)\,d_y\,(d_z\,log\,d_z) \tag{3.24}$$

if $d_x = d_y = d_z$ then the overall saving in the forward transform of magnetization vector would be at least 66.6% and it can be much more for flatter surfaces when $d_x > d_y > d_z$.

On the other hand if we use generalized FFT library such as cufft of CUDA on GPU. In this case in order to avoid the FFTs of zero padded data, we have to use 1-D FFT instead of direct 3-D FFT library of CUDA. As a result we have to perform $2D$ and $3D$ matrix transposition after "X" and "Y" transforms respectively. Transposition cost of very large data even on GPU is not negligible in overall performance of an application and it also requires additional memory. Secondly the generalized FFT libraries do not cater the specific case like the symmetries present in the input

data of micromagnetic simulations for the reduction of batches. With generalized FFT library the overall reductions that can be achieved are because of zero padded input data and it would be

$$\left[\frac{d_y}{2}\frac{d_z}{2}\right]_x + \left[(d_x)\,\frac{d_z}{2}\right]_y + \left[d_x d_y\right]_z \tag{3.25}$$

In this case the overall saving in the forward transform of magnetization vector would be 41.6% when $d_x = d_y = d_z$ with an overhead of transposition and more memory utilization.

3.7.2.3 Savings in Inverse 3-D FFT of Magnetostatic Field Vectors

Now in the inverse 3-D FFT of magnetostatic field for $X - Axis$ transform we would require d_y FFT's along $Y - Axis$ and $\frac{d_z}{2} + 1$ FFT's along $Z - Axis$ instead of d_z of size d_x. But in this case we also need a copy operation to get the missing data along $X - Axis$ before the $X - Axis$ transform as shown in figure 3.13. The complexity of transform along $X - Axis$ would be reduced to $C(d_y)(\frac{d_z}{2} + 1)\,(d_x\, log\, d_x)$ where C is copying time required to get the missing data in $X - Axis$ transform. Hence this would reduces the cost of $X - Axis$ transforms by 50% when $d_x = d_y = d_z$ plus the copying time C. For copying the missing data the same kernel for $X - Axis$ transform performs the copying operation before performing the transform, in this way we can reduce the overhead of extra kernel call for copying operation.

Similarly the complexity for $Y - Axis$ transform and for $Z - Axis$ transform would be $(\frac{d_x}{2})\,(\frac{d_z}{2} + 1)\,(d_y\, log\, d_y)$ and $C(\frac{d_y}{2}\,\frac{d_z}{2})\,(d_x\, log\, d_x)$ respectively and the savings would be almost 75% in both cases along with the copying time in case of $Z - Axis$ transform when $d_x = d_y = d_z$.

The overall reductions for the inverse (Figure-3.13), transforms would be given as:

$$copy\,(-I,-J,-K)\left[d_y\left(\frac{d_z}{2}+1\right)\right]_{x^{-1}} + \left[\frac{d_x}{2}\left(\frac{d_z}{2}+1\right)\right]_{y^{-1}} + copy\,(-K)\left[\frac{d_x}{2}\frac{d_y}{2}\right]_{z^{-1}}$$

while the overall reduction in the complexity would be

$$C(d_y)(\frac{d_z}{2}+1)\,(d_x\,log\,d_x)\;+(\frac{d_x}{2})\,(\frac{d_z}{2}+1)\,(d_y\,log\,d_y)\;+C(\frac{d_y}{2}\frac{d_z}{2})\,(d_x\,log\,d_x)$$

if $d_x = d_y = d_z$ then the overall saving in the Inverse transform of magnetization vector would be at least 66.6% plus the copying time in $X-Axis$ and $Z-Axis$ transforms and it can be much more for flatter surfaces when $d_x > d_y > d_z$. While these reductions are not being catered in generalized 3-D FFT library.

Here, the copy operations extract conjugate symmetries from the volume, plane, or strip. The order of inverse is the same as that of the forward, i.e., in the forward it was x, y, z and the inverse is also x, y, and z. Some implementations use the same FFT routines for both forward and inverse transforms. If, however, the order of the inverse is to be reversed (Figure-3.12), the number of copy operations can be reduced further to the following, resulting in around 5% decrease in simulation time (Figure-3.9):

$$\begin{aligned}&\left[d_y\left(\frac{d_x}{2}+1\right)\right]_{z^{-1}}+\left[\frac{d_z}{2}\left(\frac{d_x}{2}+1\right)\right]_{y^{-1}}\\&+copy\,(-I)\left[\frac{d_z}{2}\frac{d_y}{2}\right]_{x^{-1}}\end{aligned}\tag{3.26}$$

3.7.2.4 Savings in 3-D FFT of Demagnetization Tensor

For the demagnetizing tensor however, further reductions are possible due to the presence of odd/even symmetries within the tensor components. While the copy operation is required after each transform to get the missing data. Thus, the forward transform (Figure-3.10) can be reduced to:

$$\left[\frac{d_y}{2}\frac{d_z}{2}\right]_x+copy+\left[\left(\frac{d_x}{2}+1\right)\frac{d_z}{2}\right]_y+copy+\left[\left(\frac{d_x}{2}+1\right)\left(\frac{d_y}{2}+1\right)\right]_z+copy\tag{3.27}$$

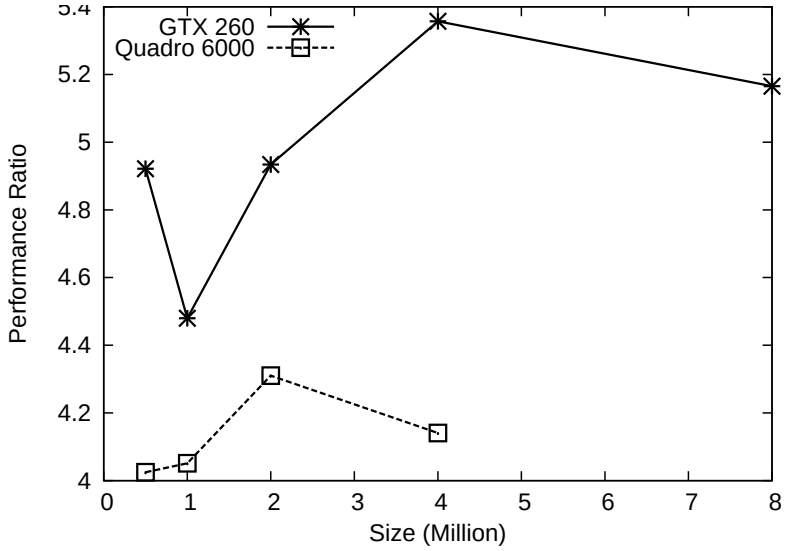

Figure 3.9. Improvement by changing order of inverse. The plot shows the ratio of improvement for two graphic cards. The presence of the trenches is due to memory coalescing factors. GTX-260, a relatively old card, performs less memory coalescing compared to the Quadro 6000. As a result, the Quadro 6000 appears to be more stable.

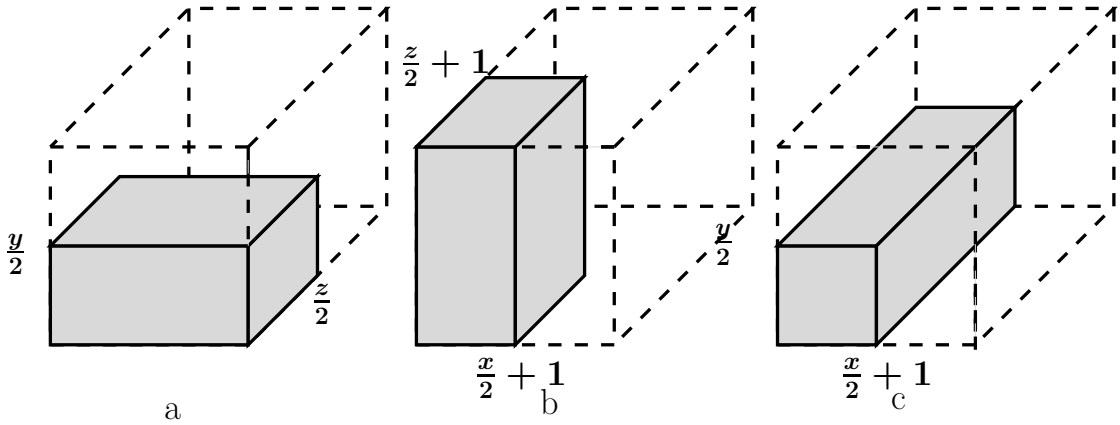

Figure 3.10. Reductions in the demagnetization tensor for the forward transform

3.8 Performance evaluation

In this section I examine the performance of my OpenCL based magnetostatic field calculation on different GPUs architecture. I first validated my results by comparing with those of OOMMF[54, 23] program developed at NIST, which is well-known

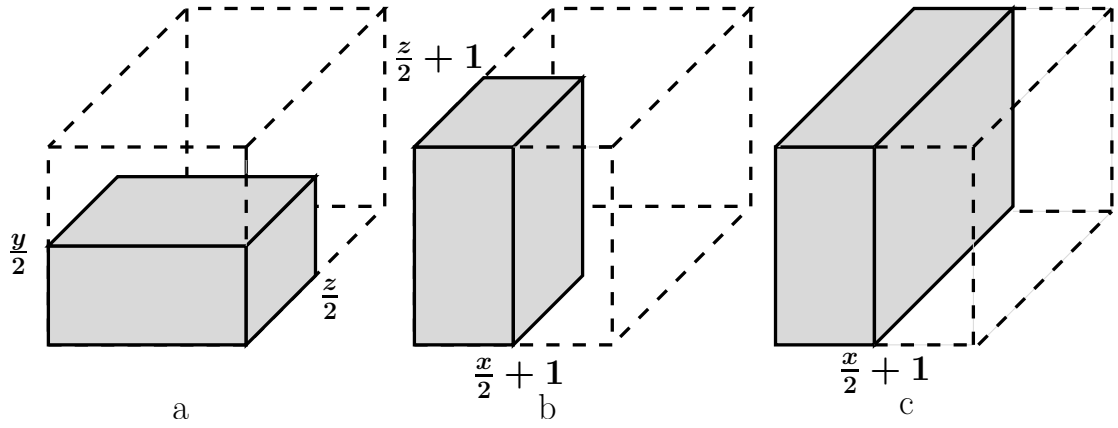

Figure 3.11. Reductions in the magnetization vector for the forward transform

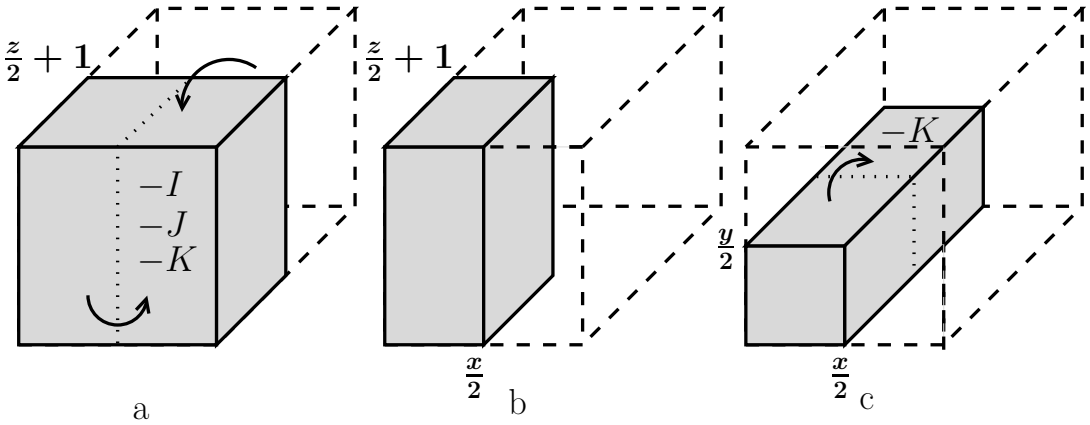

Figure 3.12. Reductions in the inverse for the magnetization vector using the same 3D FFT routines

and widely used CPU based micromagnetic solver. Then, a performance analysis is conducted by comparing the execution times of our GPU enabled code with CPU based shared memory parallel OOMMF, running on four CPU cores and with an equivalent parallel implementation on CPU. In general current generation of GPUs is better suited for single precision than double precision arithmetic. Due to GPUs architecture, having smaller number of arithmetic units, not only the double precision performance is slow, but also requires double the memory as compared to single precision [45]. This limitation of current GPUs force most of the micromagnetic simulators working on GPUs such as Mu-Max to use single precision floating point. In order to elaborate this phenomenon I have implemented my program in both

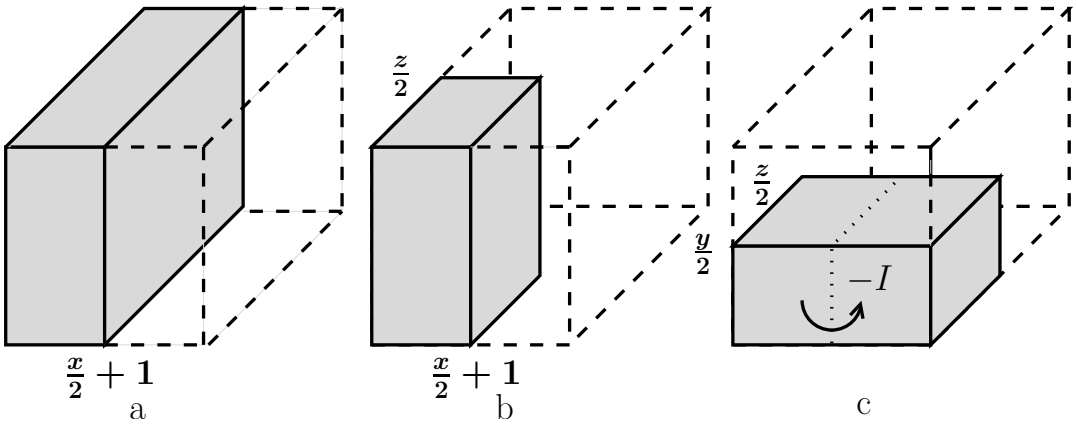

Figure 3.13. Reductions in the inverse for the magnetization vector using a different order of FFT's

double and single precision and shown the results for both the computation time and memory transfer time. Unlike most of the current GPU based micromagnetic solvers I am not using CUFFT library. Rather I have development my own OpenCL based 3D-FFT library. In this way I can easily avoid the calculation of FFTs of arrays containing only zero values, which comes as a result of zero padding in micromagnetic simulation with out using transposition. Furthermore with my own FFT library gives me the freedom to fully exploit the symmetries and as a result to avoid the redundant data copying between CPU and GPU which is a big bottleneck in GPU computing, and other optimization in magnetostatic field calculation such as memory coalescing which are specific to GPU architectures.

3.8.1 Experimental setup

I evaluated the performance of my 3-D FFT implementation for magnetostatic field calculation using two different GPU devices, NVIDIA Quadro 6000 and NVIDIA GeForce GTX 260. Table 1 shows the configuration of the GPUs and CPU used for performance evaluation. The GTX260, is a high-end graphics card with large number of cores, 216 in number and 895MB of global memory. The NVIDIA Quadro 6000, built on innovative NVIDIA fermi architecture, supports 14 microprocessors having 32 cores each, thus resulting into 448 cores in total, arranged as array of streaming multi-processors. For comparison with CPU I have used Intel Core2Quad CPU Q8400 with 2.66 GHz processor and 4GB of random memory. To test the

performance of our magnetostatic field solver I have used 2000 nm × 1000 nm × 32 nm magnetic slab, with varying the size of cubic cells, I got different input problem sizes ranges from half million to eight million spins with same x, y and z dimensions of magnetic slab for both CPU and GPU version

3.8.2 Results and discussion

In table 3.4 and 3.5, I have reported the single step computation time for magnetostatic field calculation (averaged over 5 steps) in seconds for both the OOMMF and my equivalent parallel implementation on CPU respectively against the GPU based micromagnetic solver on different GPU architectures. The computation time is reported for different input sizes from half million to eight million cells. For two different GPUs that are GTX 260 and Quadro 6000 I have shown the total computation time with and without data transfer overhead in GPU computing, along with respective speedups. Column 4 (Total Demag_m) represents the total computation time with data transfer overhead and column 3 (Total Demag) represents the computation time without data transfer overhead. Similarly column 5 and 6 shows the speedup with and without data transfer overhead respectively. Table 3.4, 3.5 and Table 3.6, 3.7 shows all the times and speedups for single and double precision floating point accuracy input data respectively.

3.8.2.1 Computation Time

In Figure 3.14 and 3.15 I have demonstrated the average computation time required in seconds by magnetostatic field in one time step of the micromagnetic simulation with double and single precision floating point accuracy respectively. I have shown the computation time both with and without memory transfer time in order to demonstrate the effect of memory transfer overhead on overall performance of the problem. The phenomenon that GPU architecture is better suited for single precision than the double precision floating point accuracy is clearly depicted by our results. The single precision implementation is almost two order of magnitude faster than double precision implementation on both GPU architectures and is same for data transfer time as the memory requirement for double precision accuracy becomes double. Figure 3.14 shows that for smaller problem sizes the OOMMF based

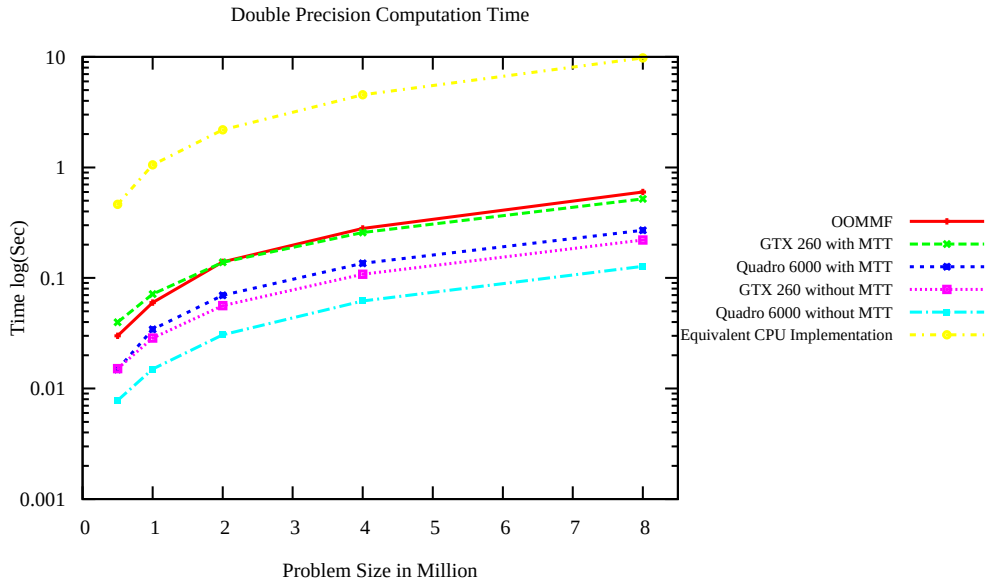

Figure 3.14. Double Precision Computation Times

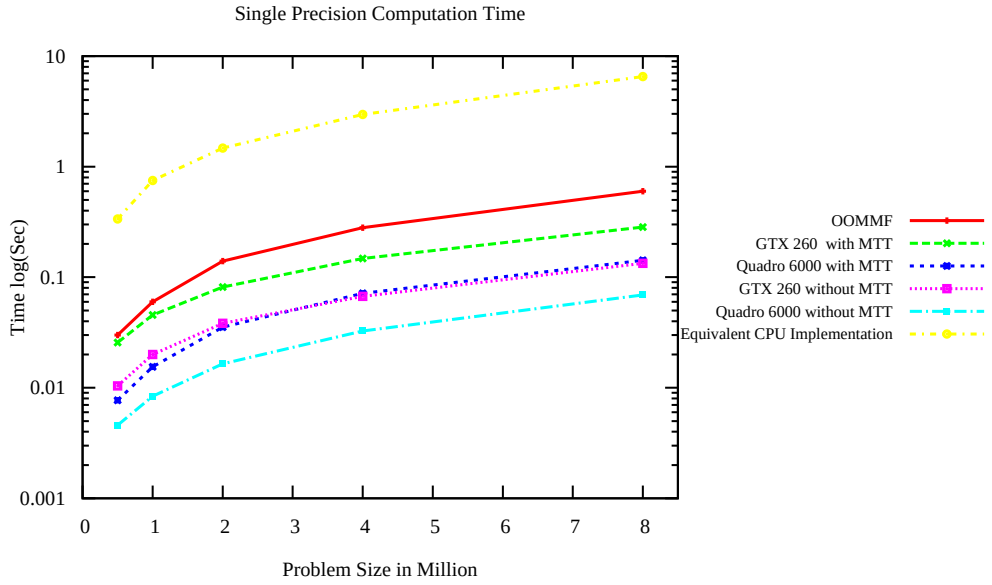

Figure 3.15. Single Precision Computation Times

CPU implementation on four cores is faster than the GTX 260. This trend can be understood with the fact, that for small problem sizes with small number of parallel

Table 3.4. Computation Time of OOMMF CPU Implementation and GPU With Single Precision GPU Implementation Against Different Problem Sizes and GPU/CPU 4-cores Speed Up Factor

	OOMMF (CPU)	GTX-260				**Quadro-6000**			
Size (million)	**Total Time(sec)**	**Total Demag**	**Total Demag_m**	**SpeedUP_m**	**SpeedUP**	**Total Demag**	**Total Demag_m**	**SpeedUP_m**	**SpeedUP**
0.5	0.03	0.011	0.026	1.169	2.882	0.005	0.008	3.892	6.538
1	0.06	0.020	0.046	1.317	3.005	0.009	0.016	3.889	7.161
2	0.14	0.039	0.082	1.721	3.651	0.017	0.036	3.966	8.506
4	0.28	0.067	0.148	1.898	4.180	0.033	0.072	3.918	8.594
8	0.6	0.135	0.mat	2.112	4.496	0.070	0.143	4.212	8.689

Table 3.5. Computation Time of Our CPU Implementation and GPU With Single Precision GPU Implementation Against Different Problem Sizes and GPU/CPU 4-cores Speed Up Factor

	Our Imp (CPU)	**GTX-260**				**Quadro-6000**			
Size (million)	**Total Time(sec)**	**Total Demag**	**Total Demag_m**	**SpeedUP_m**	**SpeedUP**	**Total Demag**	**Total Demag_m**	**SpeedUP_m**	**SpeedUP**
0.5	0.305	0.011	0.026	11.870	29.268	0.005	0.008	39.521	66.401
1	0.672	0.020	0.046	14.743	33.648	0.009	0.016	43.544	80.178
2	1.422	0.039	0.082	17.477	37.071	0.017	0.036	40.275	86.387
4	3.045	0.067	0.148	20.652	45.484	0.033	0.072	42.624	93.516
8	6.594	0.135	0.285	23.203	49.340	0.070	0.143	46.280	95.481

active threads are not fully utilizing the computation power of GPU. Secondly for small problem sizes the data transfer overhead dominates the computation time.

3.8.2.2 SpeedUp

In Figures 3.16 to 3.19 I have demonstrated the average speedup of magnetostatic field solver on different GPU architectures against CPU based shared memory parallel micromagnetic solver "OOMMF" and my equivalent parallel implementation respectively running on four cpu cores. Where my GPU based implementation shows a significant speedup factor of up to 8.6 for single precision floating point accuracy with out data transfer time and 4.2 with data transfer time on high end GPU

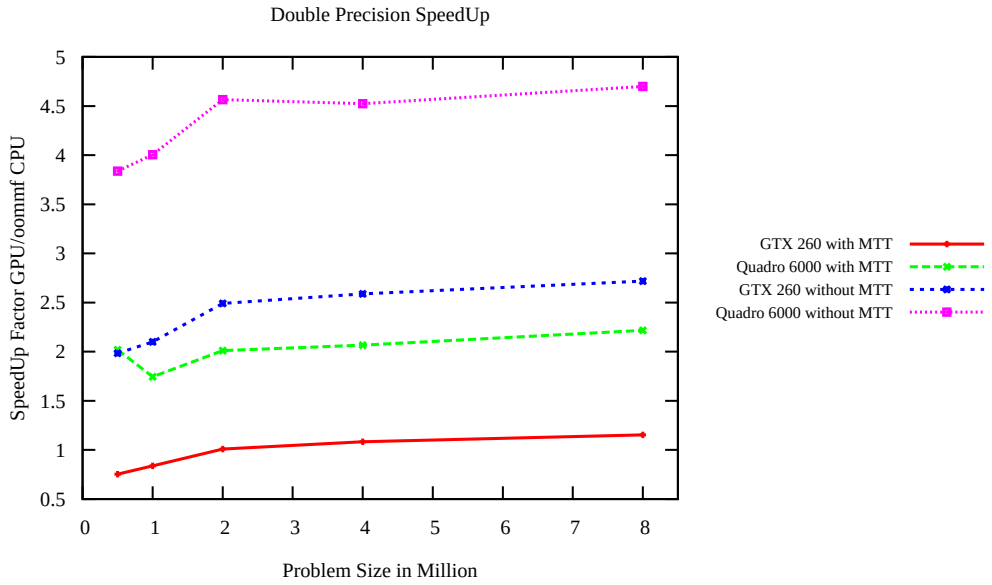

Figure 3.16. Double Precision GPU/CPU(oommf) SpeedUp where CPU code is running on 4 cores

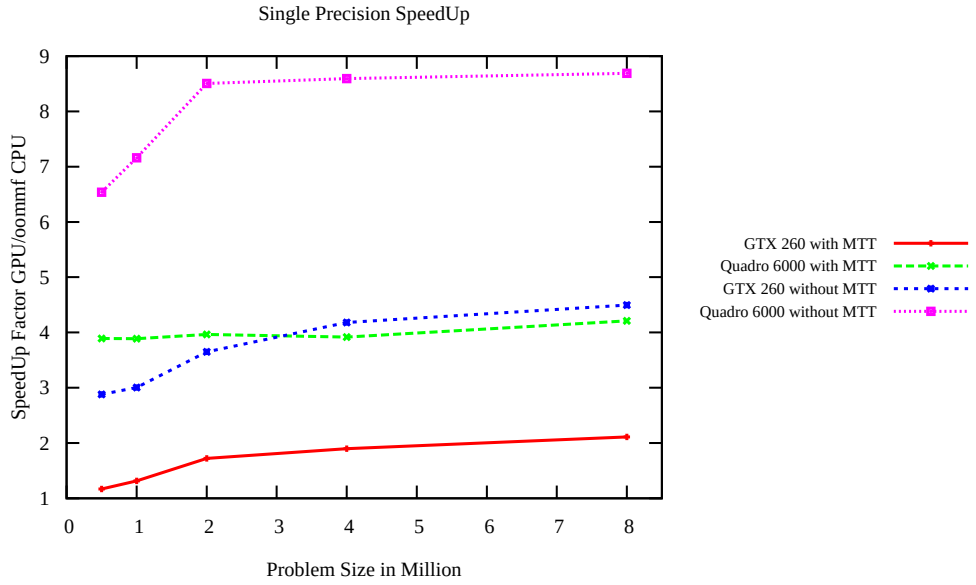

Figure 3.17. Single Precision GPU/CPU(oommf) SpeedUp where CPU code is running on 4 cores

architecture that is Nvidia Quadro-6000. On the other hand the speedup against my equivalent implementation on CPU is up to 46x and 94x with and with out data

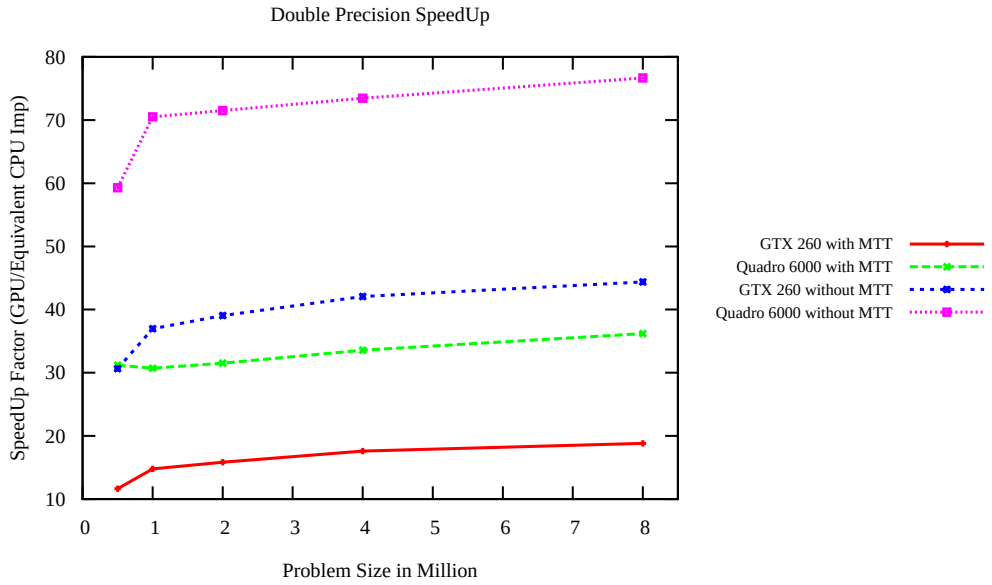

Figure 3.18. Double Precision GPU/CPU(Equivalent CPU Implementation) SpeedUp where CPU code is running on 4 cores

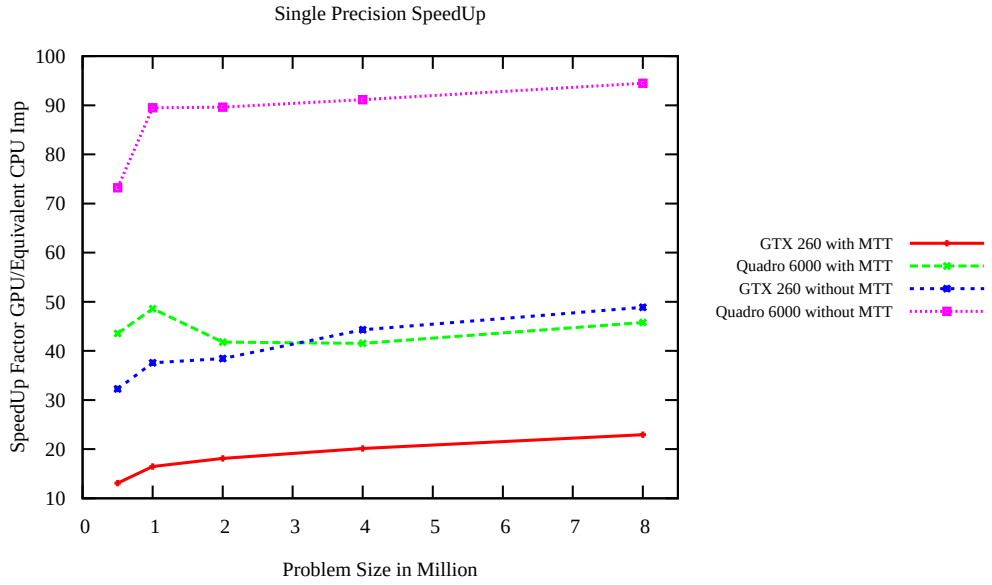

Figure 3.19. Single Precision GPU/CPU(Equivalent CPU Implementation) SpeedUp where CPU code is running on 4 cores

transfer time respectively. The reason for OOMMF for being fast on CPU against the CPU implementation is that unlike my FFT library which is currently based on

Table 3.6. Computation Time of CPU (OOMMF) and GPU With Double Precision GPU Implementation Against Different Problem Sizes and GPU/CPU 4-cores SpeedUp Factor

	OOMMF (CPU)	GTX-260 (GPU)				Quadro-6000 (GPU)			
Size (million)	Total Time(sec)	Total Demag	Total Demag_m	SpeedUP _m	SpeedUP	Total Demag	Total Demag_m	SpeedUP _m	SpeedUP
0.5	0.03	0.016	0.040	0.754	1.985	0.008	0.015	2.020	3.838
1	0.06	0.029	0.072	0.839	2.099	0.015	0.035	1.746	4.005
2	0.14	0.057	0.139	1.011	2.492	0.031	0.070	2.012	4.565
4	0.28	0.109	0.259	1.084	2.589	0.062	0.136	2.066	4.524
8	0.6	0.221	0.521	1.154	2.720	0.128	0.271	2.218	4.700

only radix-2 FFT algorithm the OOMMF program uses mix radix FFT algorithms along with other compiler based optimizations.

The speedup of GPU implementation increase with the increase in the input problem sizes. This increasing speedup trend can be understood as for larger problem sizes there are significant number of active threads to fully utilize the hundred of parallel cores on GPU secondly more active threads can also hide the latency to fetch the data from global memory on GPUs. Secondly from Figure 3.16 one can see that in either case that is without or even with memory transfer time my implementation on a single GPU is much faster than both the CPU based parallel implementations running on four cores of CPU. The two different GPUs used in my simulation differs with respect to number of parallel streaming processors and the memory bandwidth. The difference in speedup by these two GPUs is promising hence with the same implementation just by using very high end GPU one can gain significant amount of speedup on single GPU.

3.8.3 Conclusion

The graphics processing unit, which initially was designed for manipulating computer Graphics, now with the development of high level libraries and easy to use interfacing tools such as OpenCL and CUDA can be used as co-processor to speed up wide range of computation intensive applications. On the other hand in micromagnetic simulations the study of magnetization behavior at very small space and

Table 3.7. Computation Time of CPU (Our Implementation) and GPU With Double Precision GPU Implementation Against Different Problem Sizes and GPU/CPU 4-cores SpeedUp Factor

	Our Imp (CPU)	GTX-260 (GPU)				Quadro-6000 (GPU)			
Size (million)	Total Time(sec)	Total Demag	Total Demag_m	SpeedUP _m	SpeedUP	Total Demag	Total Demag_m	SpeedUP _m	SpeedUP
0.5	0.497	0.016	0.040	12.488	32.907	0.008	0.015	33.488	63.632
1	0.974	0.029	0.072	13.613	34.063	0.015	0.035	28.321	64.982
2	1.995	0.057	0.139	14.394	35.501	0.031	0.070	28.670	65.038
4	4.068	0.109	0.259	15.746	37.613	0.062	0.136	30.015	65.719
8	8.475	0.221	0.521	16.288	38.406	0.128	0.271	31.327	66.381

time scale requires lot of computational cost. Therefore parallelism becomes adequate for such type of simulations. In the recent years GPUs have provided best solution to such problems both with respect to price and performance.

In this chapter I have shown the parallel GPU implementation of magnetostatic field solver, which is the most time consuming part of micromagnetic solvers. I have demonstrated my results both for single and double precision floating point accuracy on different GPU architectures. My results show a very high speed-up factor up to 8.6x as compare to well know CPU based solver that is OOMMF and up to 94x against my equivalent CPU implementation running on four cores of CPU. Secondly while the time of writing and with best of my knowledge my implementation is the first OpenCL based magnetostatic field solver on GPUs. As unlike CUDA OpenCL targets all devices which conform to its specification therefore in future in can be used to utilized different kind of processors present in a system and for now it makes my implementation to work for different vendors' GPUs. Moreover my OpenCL based 3-D FFT library also provides a common interface to different vendors' GPUs.

In the next chapter to move further a head I have transferred this problem to multiple GPUs, which would further enhance the performance and secondly executing simulation on multiple GPUs would also solve the problem of limited on-chip memory resource on current GPUs, by utilizing the combined memory of multiple GPUs. Currently my 3-D FFT library is based on Cooley Tukey radix-2 algorithm,

in future I am planning to move towards higher radix or even mix radix FFT algorithms. Most importantly in order to achieve a very high performance I have to shift all the components of micromagnetic solver on GPU.

The publications related to this work are [44, 55, 56, 57] and the paper submitted in IEEE transactions of Parallel and Distributed Systems with the name "Fast Parallel Magnetostatic Field Solver on GPU Using Open Computing Language"

Chapter 4

Magnetostatic field computation on multiple GPUs

4.1 Why parallel GPUs

In chapter 3 I have discussed the highly optimized implementation of magnetostatic field computation on many core architecture that is GPU. The natural next step to further increase the level of parallelism and performance is the use of multiple GPUs. In micromagnetic simulation due to very fine space and time discretization normally the input problem sizes are very large to accommodate as a whole on a current generation of GPUs' memory [58]. This problem can be handled by using the combined resources of multiple GPUs.

The thread management on each GPU is done automatically at hardware level. While when we talk about multiple GPUs and want to compute the problem in parallel on multiple GPUs then we require additional management regarding the synchronization and communication overhead while decomposing the input problem on multiple GPUs. In this way along with solving the problem of limited memory resources we can also achieve high performance by utilizing the combined computational resources of multiple GPUs.

In this chapter I will discuss the parallel magnetostatic field solver on multiple GPUs.

4.1.1 Limitations of single GPU implementation

In the recent years the use of GPUs for general purpose computation has increased dramatically due to the reason such as

- Performance
- Economical
- Memory Bandwidth
- Architecture

which I have discussed in detail in section 2.3.1. However as compared to CPU main memory, the memory on GPUs is normally limited. For many memory hungry applications such as [59, 60, 61] which requires a lot of memory along with computation time, the limited memory can be a bottleneck for GPUs. Like in our case in micromagnetic simulations due to very fine space and time discretization makes it very large problem both with respect to computation and memory consumption point of view. Therefore moving towards multiple GPUs is not only required because of improving the computational time but also necessary to overcome the limited memory problem on single GPU.

4.1.2 Multiple GPUs advantages

With multiple GPUs, the problem of limited memory can be tractable by divide and conquer approach that is by dividing the input data and the computation among available GPUs. In short with multiple GPUs implementation we can have following benefits

- Overcomes the limited memory bottleneck on single GPU.
- Further increase the performance by parallel execution on multiple GPUs.

while shifting on to multiple GPUs along with the above mentioned benefits, there also arises some problems which we have to consider such as

- The Data dependency problem.

- The communication overhead.

Therefore the performance of the problem on multiple GPUs heavily depends on these two factors which one have to consider. For efficient use of multiple GPUs the computation time required by the input problem must be much greater than the communication overhead time.

4.2 Different architectural approaches

With respect to computation GPUs are very powerful, but when the input problem size becomes becomes very large both from the point of view of memory consumption and computation time one wish to compute it on multiple GPUs. Multiple GPUs can have either of the following two architectural approaches

4.2.1 Shared system GPUs

In case of shared system GPUs the multiple GPUs are connected to same system through PCI/AGP slots and shares the same CPU RAM for communication with each other. Along with multiple cards on different PCI/AGP slots there are some cards like GTX 295 are dual core GPUs but appears as two separate GPUs. On single core machine multiple threads can be used to handle multiple GPUs while on the other hand it is advantageous to have multi-core CPU so each core handles separate GPU.

Secondly with respect to communication cost in case of shared system multiple GPUs as all the GPUs are physically present on single system therefore the communication between them took place through PCI buses. Shared system GPUs' configuration is depicted in figure 4.1

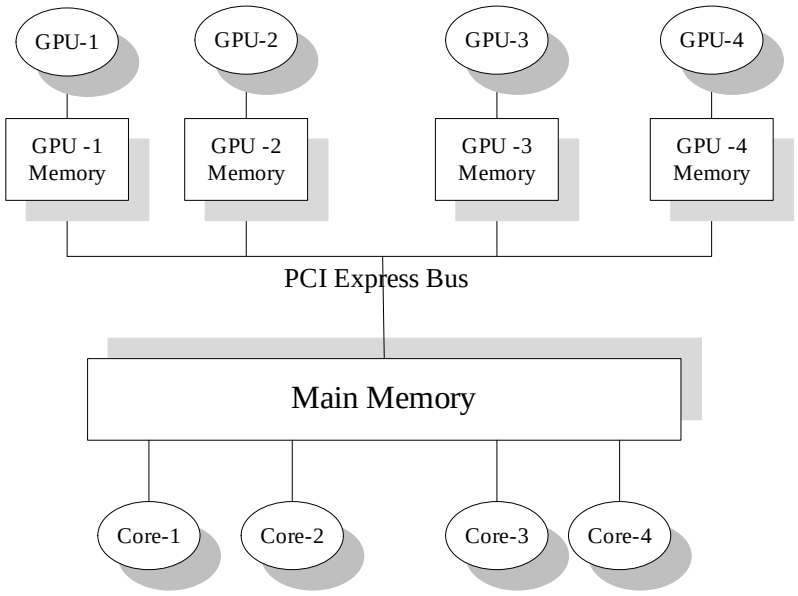

Figure 4.1. Shared System Multiple GPUs on multicore CPU system and sharing same CPU memory

4.2.2 Distributed system GPUs

While on the other hand in case of distributed system multiple GPUs' configuration multiple GPUs are present on separate CPU node and communicate to each other via some underlying communication network similar like the cluster computing. The difference between the CPU cluster and the distributed system GPUs is that in the later case for the communication between two GPUs we require two different levels of communication. First the data transfer between the GPU memory to CPU and then from CPU memory to second node through underlying communication network.

In parallel computing the performance of the system depends on

- Degree of parallelism of the code.
- Number of parallel threads.
- The communication cost among the parallel threads.

In case of distributed system GPUs as we can have very large number of GPUs which can run large number of parallel threads but the overall performance gain heavily depends on the type of underlying communication network being used. Like

in case of infiniBand network we can reduce this cost to some extend but along with many other factors the financial cost of infiniBand is also a hurdle in most of the cases. Distributed system GPUs' configuration is depicted in figure 4.2

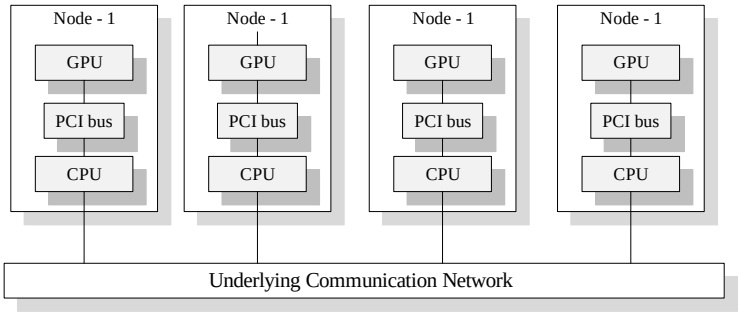

Figure 4.2. Distributed System Multiple GPUs where each CPU node contains one GPU and connected to each other through underlying communication network

4.3 Magnetostatic field computation on multiple GPUs

As discussed before that in micromagnetic simulations due to very fine space and time discretization of magnetic body the input problem sizes are normally very larger to accommodate on single GPU memory. In particular the magnetostatic field computation which is dominated by FFT computation is most time and memory consuming part and is iteratively computed in each time step.

In magnetostatic field computation via discrete convolution method whose expanded form is given in equation 4.1 is

$$\begin{aligned}
\tilde{\mathbf{H}}_{x(i,j,k)} &= \tilde{\mathbf{N}}_{xx'(i,j,k)}\tilde{\mathbf{M}}_{x(i,j,k)} + \tilde{\mathbf{N}}_{xy'(i,j,k)}\tilde{\mathbf{M}}_{y(i,j,k)} + \tilde{\mathbf{N}}_{xz'(i,j,k)}\tilde{\mathbf{M}}_{z(i,j,k)} \\
\tilde{\mathbf{H}}_{y(i,j,k)} &= \tilde{\mathbf{N}}_{yx'(i,j,k)}\tilde{\mathbf{M}}_{x(i,j,k)} + \tilde{\mathbf{N}}_{yy'(i,j,k)}\tilde{\mathbf{M}}_{y(i,j,k)} + \tilde{\mathbf{N}}_{yz'(i,j,k)}\tilde{\mathbf{M}}_{z(i,j,k)} \\
\tilde{\mathbf{H}}_{z(i,j,k)} &= \tilde{\mathbf{N}}_{zx'(i,j,k)}\tilde{\mathbf{M}}_{x(i,j,k)} + \tilde{\mathbf{N}}_{zy'(i,j,k)}\tilde{\mathbf{M}}_{y(i,j,k)} + \tilde{\mathbf{N}}_{zz'(i,j,k)}\tilde{\mathbf{M}}_{z(i,j,k)}
\end{aligned} \tag{4.1}$$

where the FFT quantities are with tilda sign. From equation 4.1 we can see that for any point (i, j, k) in 3D discretized magnetic body the computation of demagnetizing field H_m along each x, y and z direction in cartesian coordinates requires in total twelve FFTs. The six FFTs belongs to demagnetizing tensors which are computed once and stored on memory. From rest of six FFTs, three related to

magnetization vectors, ($Mvec$) namely $\mathbf{M_x}$, $\mathbf{M_y}$ and $\mathbf{M_z}$ and three inverse FFTs of the ($Hvec$) components $\mathbf{H_x}$, $\mathbf{H_y}$ and $\mathbf{H_z}$ and these have to be computed for each time step.

Now we have two ways to solve this problem on GPU. The first way which is being followed by all current GPU implementations is that we move all the data that is the quantities with tidle sing in equation 4.1 to GPU and performs the whole convolution operation on GPU. But the problem in this method is limited GPU memory. In this way we can handle only a problem sizes up to some limited sizes. For example like in our case where I am using GTX 260 card which has a global memory of one gigabyte. With one GB (gigabyte) memory we can handle maximum of four million discretized cells in a magnetic body with double precision floating point accuracy and maximum of eight million cells in case of single precision floating point accuracy. This method is adopted by current implementation because in this way they transfer the whole data at the beginning of the simulation to the GPU memory and copy back only the final results at the end. By doing so they do not need intermediate data transfer between GPU and CPU and vice versa, which is most time consuming part. Therefore this method there is a tradeoff between the input problem sizes and the computation time.

The second way in which we can solve this convolution problem on GPU is divide the problem in to components and perform FFT on each component one by one. As discuss above we have to perform the FFT of twelve different components. With this method even on a single GPU the input problem size can be roughly twelve times the size which we can handle with the first method discussed above.

I am using this method for my implementation of magnetostatic field computation. This method helps to handle wide range of input problem sizes even on medium range graphic cards like Nvidia's GTX-260. In order to mitigate the extra data communication overhead between CPU and GPU and vice versa. I have developed my own multi dimensional FFT library on GPUs, which made it possible to fully exploit the zero padded input data without transposition and symmetries inherent in the field calculation. As a result the complexity of overall system reduced significantly compared to current GPU based solvers. Moreover it also provides a common interface for different vendors' GPUs. In order to fully utilize the GPUs parallel architecture my solver handles many hardware specific technicalities such

as coalesced memory access, data transfer overhead between GPU and CPU, GPU global memory utilization, arithmetic computation, batch execution etc. The different optimization strategies are discussed in detail in section 3.7 to mitigate the overhead of CPU to GPU data transfer time.

So the natural next step is the use of multiple GPUs to further increase the computation power and to avoid the limited memory problem on single GPU by exploiting the combined resources of multiple GPUs.

4.3.1 Execution model on multiple GPUs

OpenCL kernel executes on the specified devices defined in the environments known as context. The context is considered as a package containing different resources such as devices to be used to run the OpenCL kernel, kernels, program objects and the memory objects [62].

The data structure known as command queue is created on the host which holds the commands by host to be executed on the devices defined in the context. For multiple GPUs we have two different approaches to create and manage the context and the GPU devices related to it.

4.3.1.1 Single Context Multiple Devices

Single context multiple device approach is considered as standard approach for shared system multiple GPUs in OpenCL. In this approach the OpenCL objects such as memory object, kernel, program etc are shared among all the devices belonging to that specific context. On multiple GPUs there is no direct mechanism to transfer data between them, the copy command only exist to copy data from device (GPU) to host (CPU) or from host to device. Figure 4.3 depicts the copy mechanism between two devices belonging to same context which shows the intermediary copy to host for transferring data between two GPUs.

In my implementation as multiple GPUs are present on single system that is why I am using this approach in execution model, which reduces the extra communication cost on underlying network to transfer data among multiple GPUs which is explained in next section.

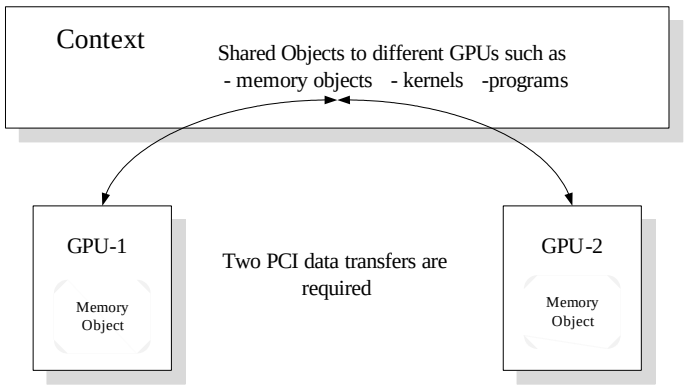

Figure 4.3. The data communication among multiple GPUs using single context multiple devices approach

4.3.1.2 Multiple Contexts Multiple Devices

Multiple contexts multiple devices approach is used where there exist distributed system multiple GPUs model. In this approach each device has its own context and the objects created in that context are only associated to it. In this approach along with PCI data transfer between host and device additional communication is required between the communicating GPUs using host based libraries such as pthreads or MPI on underlying communication network. This scenario is depicted in figure 4.4. Therefore the performance gain in this scenario heavily depends on the communication network being used to connect the different nodes.

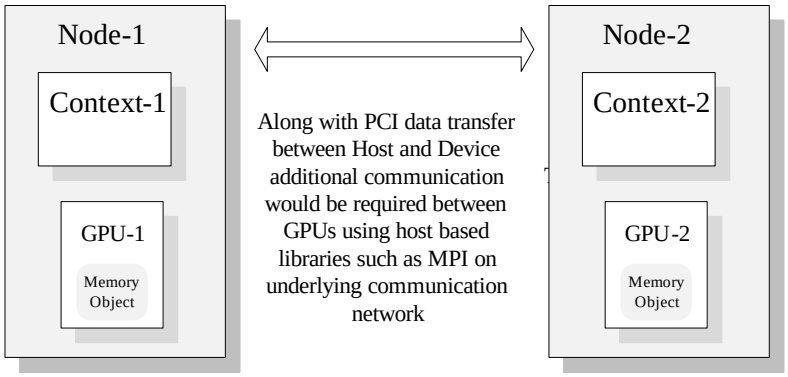

Figure 4.4. The data communication among multiple GPUs using multiple context multiple devices approach

4.3.2 Work division on multiple GPUs

The first and the most important consideration for distributing the workload among multiple GPUs is the data transfer overhead. If the input data is not divided properly the performance of multiple GPUs may even get worse than single GPU implementation just because of the slow inter GPUs communication as discussed in section 4.3.1.1.

There are two approaches for designing multiple GPUs program with respect to input data.

- Keep the redundant copy of all input data and set the global offsets for indexing.
- Divide the input data into subsets and index them locally in subset.

In my case where I am trying to save the memory for very large input problem sizes first case is not feasible as it requires more memory and also the more data transfer time. Therefore in my implementation I have adopted the second approach for dividing data among multiple GPUs which saves the extra memory along with the extra data transfer time.

Secondly with respect to 3D input data, there are different possibilities to divide the data among multiple GPUs. Figure 4.5 depicts my data division scheme on multiple GPUs which requires no data transfer for 2D FFT and for the for the third dimensional FFT it requires data transfer among participating GPUs. This approach is similar to slab decomposition used by a popular FFT library on CPU that is FFTW on distributed memory systems [63] except here I am not using the global transposition for the third dimensional FFT. There are different reasons for not using the global transposition on multiple GPUs for the third dimensional FFT such as

- For making the data local to GPUs the transposition would also require a data transfer among multiple GPUs.
- Extra kernel invocation for transposition.
- Extra computation time for transposition.

Therefore to overcome these problems we can access the data for the third dimensional FFT by sharing the memory objects on different devices which is discussed in next section.

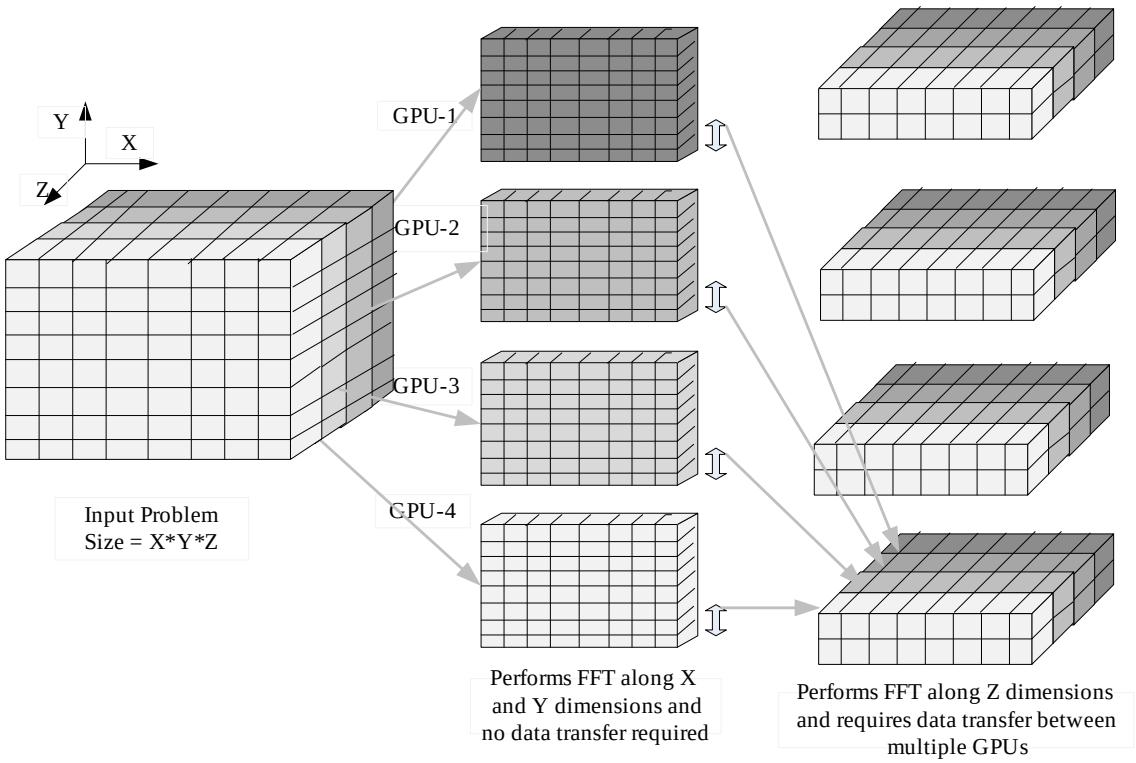

Figure 4.5. Data division among multiple GPUs

4.3.3 Sharing memory objects on multiple GPUs

As discussed in section 4.3.1.1 when we create a single context for multiple devices the different OpenCL objects such as memory object, kernel, program etc are shared among all the devices belonging to that specific context. Courtesy to shared memory object we can avoid the global transposition of input data to perform the third dimensional FFT locally. With shared memory objects the data required by a GPU is directly accessed from the memory location of the specific GPU with no extra CPU intervention.

4.3.4 Performance evaluation on multiple GPUs

In this section I would examine the performance of my OpenCL based magnetostatic field calculation on shared system multiple GPUs architecture. Performance analysis is conducted by comparing the execution times of single GPU based implementation of magnetostatic field solver against the implementation running on four GPUs in parallel. The use of multiple GPUs not only increases the performance but also mitigates the limited memory problem on GPUs by utilizing the combined resources on multiple GPUs.

4.3.4.1 Experimental setup

The graphic cards which I am using for my implementation are Nvidia Geforce GTX 295, where each GTX 295 contains two GPU cores. I am using two of these cards on two different PCI slots hence total of four GPUs in parallel. Table 4.1 shows the configuration of the GTX 295 which is used for performance evaluation. To test the performance of our magnetostatic field solver we used 2000 nm × 1000 nm × 32 nm magnetic slab, with varying the size of cubic cells, we got different input problem sizes ranges from half million to sixteen million spins with same x, y and z dimensions of magnetic slab for both single and multiple GPU implementations.

4.3.4.2 Computation Time

Figure 4.6 shows the average computation time of magnetostatic field solver on single GPU against the four GPUs in parallel for different input problem sizes. Theoretically the speedup should be four times the single GPU but as we can see from the results in figure 4.7 that actual speedup is not exactly four times the single

Architecture Details	**Nvidia GForce GTX 295**
Total Processing Cores	30 (Per GPU core)
Micro Processors	240 (Per GPU core)
Core Clock Rate (MHz)	576
GFLOPS	1788
Mem. Bandwidth (GB/s)	2*111.9

Table 4.1. Architecture details of GForce GTX 295

GPU implementation. The reason for this is the extra communication overhead to transfer the data among multiple GPUs. As discussed above in section 4.5 that data is equally divided on multiple GPUs. For performing the X and Y transform we do not need to transfer data among multiple GPUs but for the third dimensional FFT the data is no more available locally on the GPUs, therefore data transfer is required. The data transfer procedure is discussed in section 4.3.1.1.

The speedup is almost constant for different sizes, except very small input problem sizes due to the overhead related to each data transfer [64]. It is better to transfer data in large segments rather than transferring in small batches to reduce the data transfer overhead.

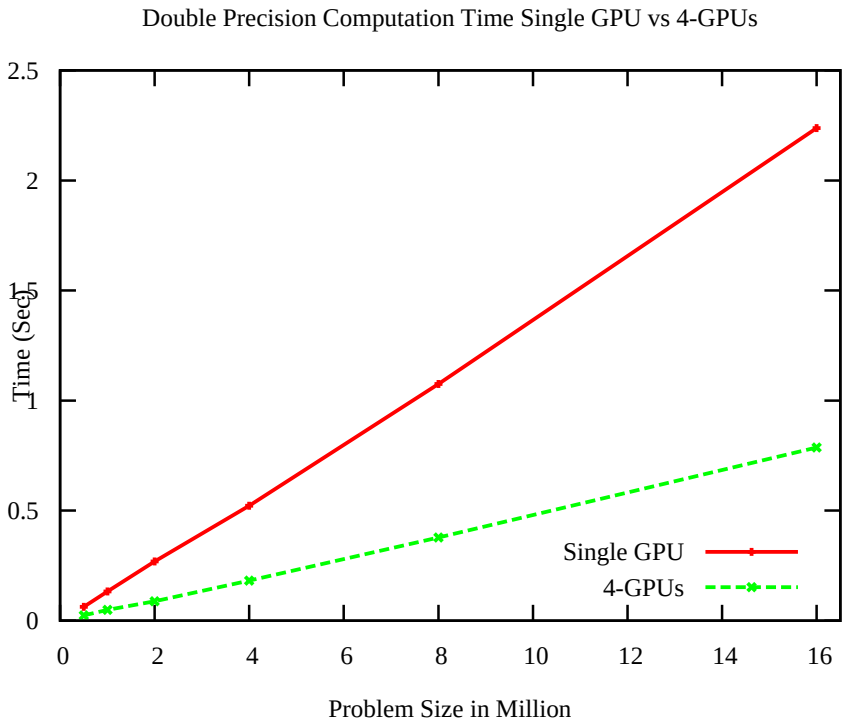

Figure 4.6. Magnetostatic field computation time on single GPU against four GPUs in parallel for different input problem sizes

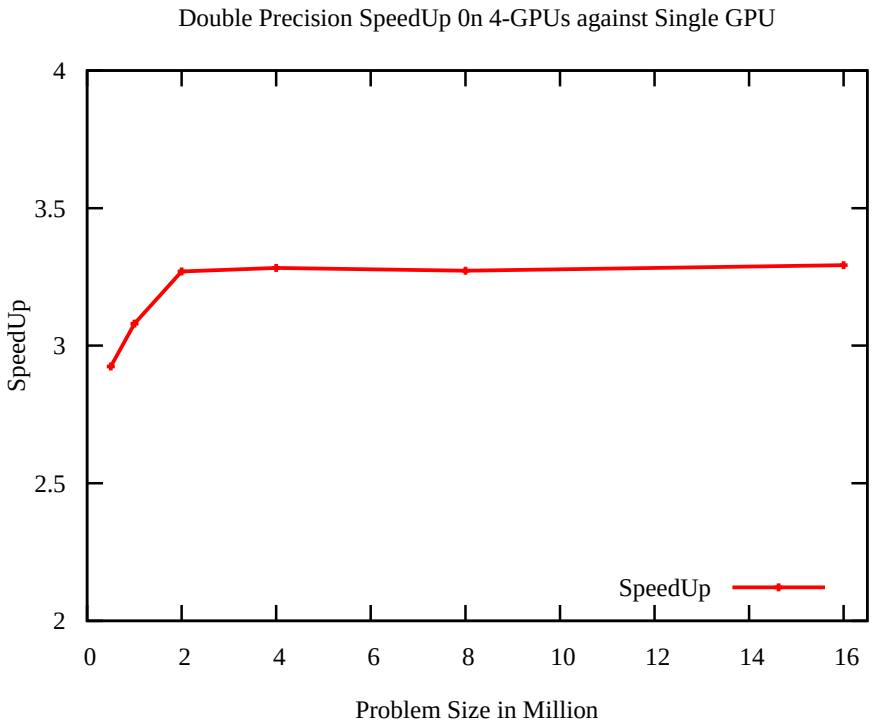

Figure 4.7. The speedup of Magnetostatic field computation on four parallel GPUs against single GPU implementation for different input problem sizes

4.3.5 Conclusion

In chapter 3 I have discussed the implementation of magnetostatic field solver on single GPU and shown the speedup against equivalent CPU implementation for different input problem sizes. I have used different optimization techniques both with respect to our input problem and the GPU hardware architecture. Normally in micromagnetic simulations due to very fine space and time discretization the input problem sizes are very large. On the other hand on most of the GPUs have little memory resources to accommodate the very large input problem sizes. Therefore the natural next step to further increase the performance of magnetostatic field solver on GPUs and to mitigate the limited memory problem is the use of combined resources of multiple GPUs in parallel.

In this chapter I have discussed the different issues and techniques related to the multiple GPUs implementation. I have shown the implementation of magnetostatic

field solver on multiple GPUs in parallel and the speedup against single GPU implementation. In this way we can handle very large input problem sizes by utilizing the memory resources on different GPUs along with computation speedup.

The publication on this work that is "Fast Parallel Magnetostatic Field Solver on Multiple GPUs " is under progress and hopefully would be ready soon to submit in well reputed journal.

Part III

Queue Management on GPUs

Chapter 5

Fast parallel sorting algorithms on GPUs

5.1 Introduction

Sorting algorithms have been studied extensively since past three decades. Their uses are found in many applications including real-time systems, operating systems, and discrete event simulations. In most cases, the efficiency of an application itself depends on usage of a sorting algorithm. Lately, the usage of graphic cards for general purpose computing has again revisited sorting algorithms.

In this chapter I would present a novel Butterfly Network Sorting algorithm (BNS) for sorting large data sets on GPUs. A minimal version of the algorithm Min-Max Butterfly is also shown for searching minimum and maximum values in data. Both algorithms are implemented on GPUs using OpenCL exploiting data parallelism model. Results obtained on different GPU architectures show better performance of butterfly sorting in terms of sorting time and rate. The comparison of butterfly sorting with other algorithms:bitonic, odd-even and rank sort show significant speedup improvements on Nvidia Quadro-6000 GPU with relatively better sorting time and rate.

5.2 Related Work

Sorting is one of the most extensively studied algorithms since well over three decades. Likewise, there is abundant literature on the topic. Since it is not possible to mention all previous sorting algorithms in this section, I am presenting an occurrence of parallelism in literature only with respect to GPUs in this section. An overview of parallel sorting algorithms is given in [65].

A quick-sort implementation on GPU using CUDA is considered in [66]. The quick-sort algorithm discussed in [66] works in two steps, creation of sub-sequences and assigning threads to the sub-sequences generated in first step. Their algorithm works in divide and conquer fashion on left-right sequences formation in accordance to the current value, greater or smaller than the value of pivot. The results in [66] show better performance of quick sort over bitonic and radix sort for large sequences with complexity of $O(nlog(n))$ and $O(n^2)$ for the worst case. For smaller sequences they suggested bitonic sort. A GPU implementation of merge sort and radix sort is presented in [67]. In this case, the radix sort divides the sequence of $N-items$ into N/P blocks. In next phase, in order to maximize coherence of scatters and minimize it to global memory, every sequence then is sorted by radix sort exploiting shared memory on the chip. The merge sort algorithm discussed in [67] adopts same divide and conquer approach where the complete sequence is divided into p equally sized tiles. All tiles are sorted in parallel with p thread blocks using odd-even sort, and then merged together using merge-sort conventions on a tree of $logp$ depth.

An adaptive bitonic sorting algorithm is shown in [68]. Their implementation achieves optimal complexity of $O(nlog(n)/p)$ for sorting n numbers on p streaming processors. A GPU implementation of bitonic sort is discussed in [69] and CUDA based in-place bitonic sort is implemented in [70]. An overview of sorting on queues is covered in [71] focusing mainly on traffic simulations for studying the behavior of transport agents in large groups. A parallel implementation of odd-even sort suggested in [24] shows that parallelism can be introduced at each stage only internally i.e. at compare-exchange process but not stage by stage meaning that no two stages can be executed in parallel as output at any stage s_i is input for subsequent stage s_{i+1}. Same holds true for both min-max butterfly and full butterfly sorting where consecutive two stages can not be executed in parallel.

5.3 Butterfly structure

I have considered 2×2 butterfly acting as compare-exchange circuit for both min-max butterfly and full butterfly sorting. The input values at the inner upper and lower wings of the butterfly are compared and exchanged if needed and places the output values at outer upper and lower wing of the butterfly as shown in figure 5.1. The detailed working of both min-max and full butterfly sorting is given in the next sections.

5.3.1 Min-Max Butterfly

The min-max butterfly finds minimum and maximum in large volume of data using butterfly compare-exchange circuit. Min-max butterfly for searching minimum and maximum in N size data has total of $log_2 N$ stages. Complexity in terms of butterflies (comparators) is $(N/2)log_2 N$ butterflies where $N/2$ are number of butterflies in each stage. An example diagram of length 8 min-max butterfly is shown in Figure 5.1. Here $x(0), x(1)...x(7)$ can be any random values. At each stage $N/2$ butterflies are carried out in parallel where each butterfly fetches two values, Pos_{start} and Pos_{end}, from queue and then compares these values to be placed either at its upper or lower wing of butterfly accordingly as shown in the algorithm below. After successful complete run of the algorithm in this case minimum and maximum values, 0 and 7, are output at$x(0) and x(7)$ respectively. The min-max algorithm is carried out stage by stage with parallelism introduced by executing butterflies in parallel inside any stage.

In addition to finding minimum and maximum in data, the minmax butterfly does complete sorting in special cases where input data is completely in descending order and vice versa.

The min-max butterfly algorithm works as follows

5.3.2 Full Butterfly Sorting

In this section I would explain the novel butterfly sorting algorithm on GPUs. The butterfly sort orders input data following any distribution type:uniform, random,

```
input  : data_random,size
output: data_sorted
begin
    for x_out ← 1 to log2(size) do
        PowerX = radix^x_out;
        do parallel T
            yIndex = t/ (PowerX/radix);
            kIndex = t% (PowerX/radix);
            Pos_start = kIndex + yIndex × PowerX;
            Pos_end = kIndex + yIndex × PowerX + PowerX/radix;
            if Pos_start > Pos_end then
                swap(Pos_start,Pos_end)
            end
        end
    end
end
```

Algorithm 1: Min/Max Butterfly Sorting Network

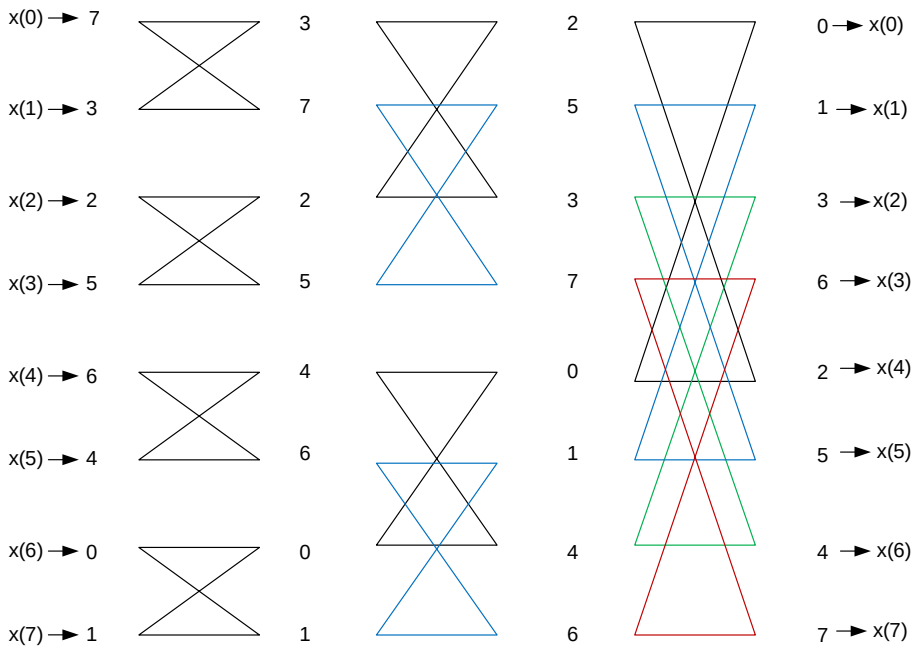

Figure 5.1. Min-Max Butterfly

exponential etc. Like min-max butterfly, in full butterfly sort the number of butterflies in any stage are constant i.e. $N/2$. For complexity in terms of total butterflies, we first find total number of stages which is given by the following formula.

$$Total_{stages} = log_2 N + \sum_{i=1}^{log_z N - 1} i \tag{5.1}$$

$$Total_{butterflies} = N/2 \times Total_{stages} \tag{5.2}$$

In equation 5.1, $log_2 N$ are total number of out-kernals represented by the first *do–parallel* block of the algorithm where as $\sum_{i=1}^{log_z N-1} i$ are total number of in-kernels represented by second *do–parallel* block in the algorithm. Figure 5.2 shows an example of length 16 full butterfly network. The big and the most prominent difference between the bitonic sort and the butterfly sort is that of bitonic sequence that is in bitonic sort before sorting an arbitrary sequence it must be converted in to bitonic sequence. A bitonic sequence is a sequence which either monotonically increases or decreases, reaches a single maximum or minimum, and then after that

maximum or minimum value it again monotonically increases or decreases. On the other hand in butterfly sort we do not need to create any sequences before the sorting.

The butterfly for the full sorting network is given as follows:

```
input  : data_random,size
output: data_sorted
begin
    for x_out ← 1 to log2(size) do
        PowerX = radix^x_out;
        do parallel T
            yIndex = t/(PowerX/radix);
            kIndex = t%(PowerX/radix);
            Pos_start = kIndex + yIndex × PowerX;
            Pos_end = PowerX − kIndex − 1 + yIndex × PowerX;
            if Pos_start > Pos_end then
                swap(Pos_start,Pos_end)
            end
        end
        if x > 1 then
            for x_in ← x to 1 do
                PowerX = radix^x_in;
                do parallel T
                    yIndex = t/(PowerX/radix);
                    kIndex = t%(PowerX/radix);
                    Pos_start = kIndex + yIndex × PowerX;
                    Pos_end = kIndex + yIndex × PowerX + PowerX/radix;
                    if Pos_start > Pos_end then
                        swap(Pos_start,Pos_end)
                    end
                end
            end
        end
    end
end
```

Algorithm 2: Full Butterfly Sorting

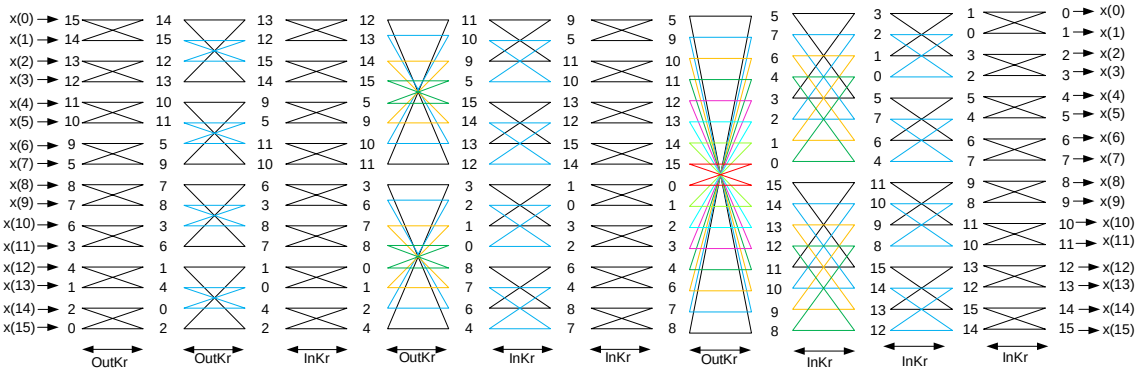

Figure 5.2. Butterfly Sorting

5.4 Performance Analysis

Performance of the sorting algorithms discussed here is evaluated both on CPU and GPUs considering their sequential and parallel implementations in terms of sorting time, sorting rate and speedup.

5.4.1 Experimental Setup

All simulations are carried out in OpenCL 1.2 and standard C compiler for different queue sizes in the power of 2. Input data of type float, is taken from a random number generator with size in the range of 2^{10} to 2^{25}. Variable declaration/ initializations, random number generators and other memory reads/writes to/from queues are mainly limited to CPU in host program. Actual sorting, butterfly computation, is carried out on GPU in kernel code. Hardware architectures used for simulations are Nvidia-Quadro6000, GeForce GTX260 and GeForce GT320M for parallel implementation and Intel Core2Quad CPU Q8400 for serial implementation.

5.4.2 Results

5.4.2.1 Sorting Time

Sorting time of the algorithm is recorded as real time in seconds and is the time spent by the algorithm only for sorting data and excludes any other time spent in variable initialization, memory read/write and contention times etc. Sorting times for min-max butterfly and full-butterfly sorting on different GPU and CPU architectures are depicted in Figures 5.3 and 5.4 respectively. Performance is improved by exploiting high parallelism inside any stage of the algorithm. Sorting time and rate values for full butterfly sorting are relatively better than bitonic sort, odd even sort and rank sort as shown in [24].

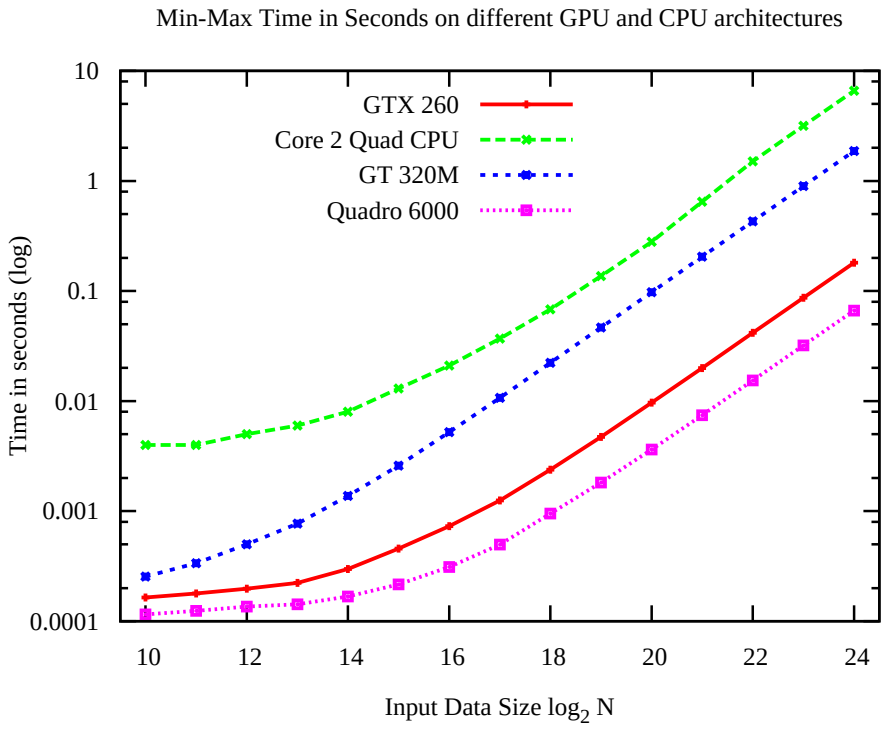

Figure 5.3. Sorting Time Min-Max Butterfly

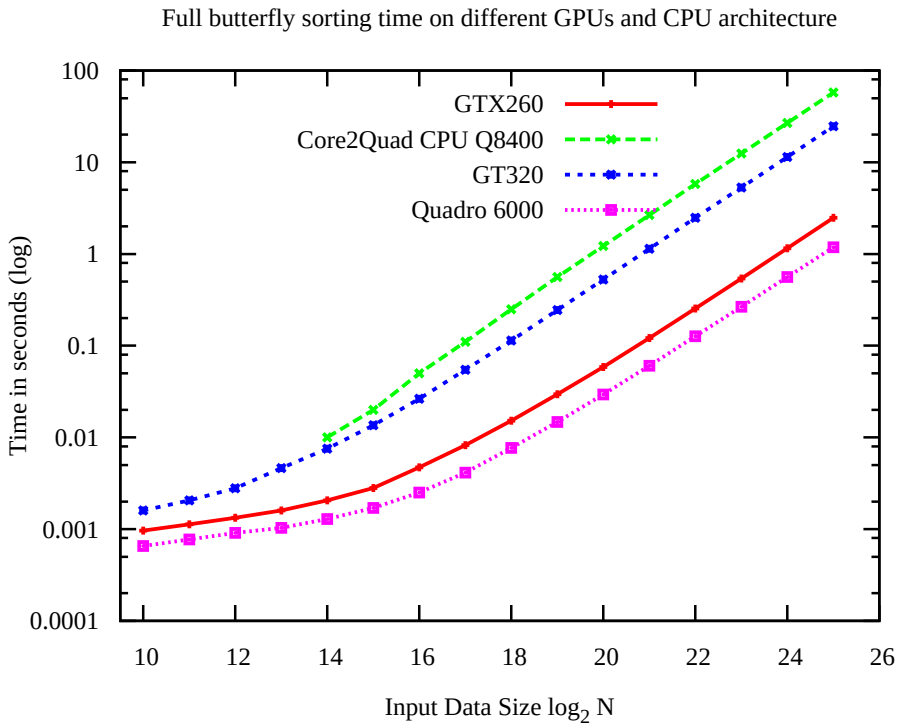

Figure 5.4. Sorting Time Full Butterfly Sort

5.4.2.2 Sorting Rate

Sorting rate is the ratio of queue size to sorting time. Sorting rates for bitonic, odd/even and rank reported in[8] and are used only for comparisons with sorting rates of minmax butterfly and full butterfly. Our results for sorting rates, Figures 5.5 and 5.6, of min-max and full butterfly sort show better performance than all the three algorithms.

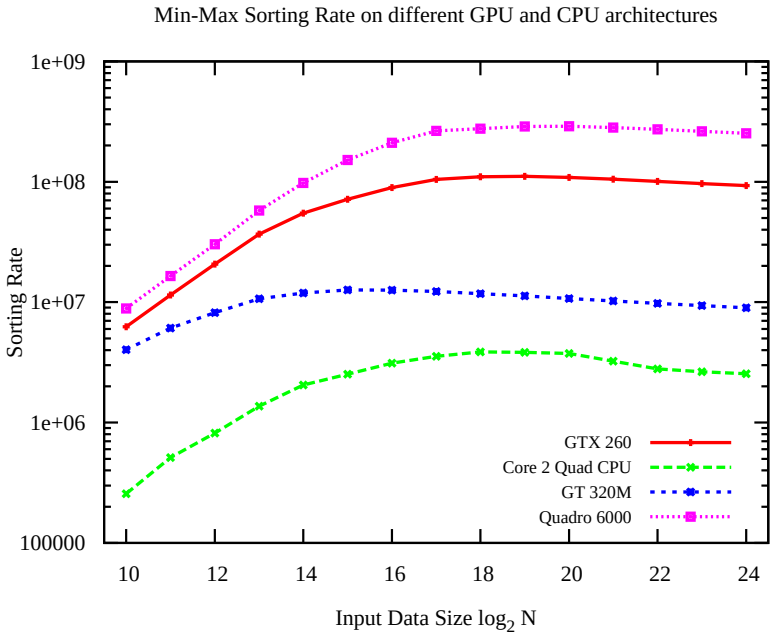

Figure 5.5. Sorting Rate Min-Max Butterfly

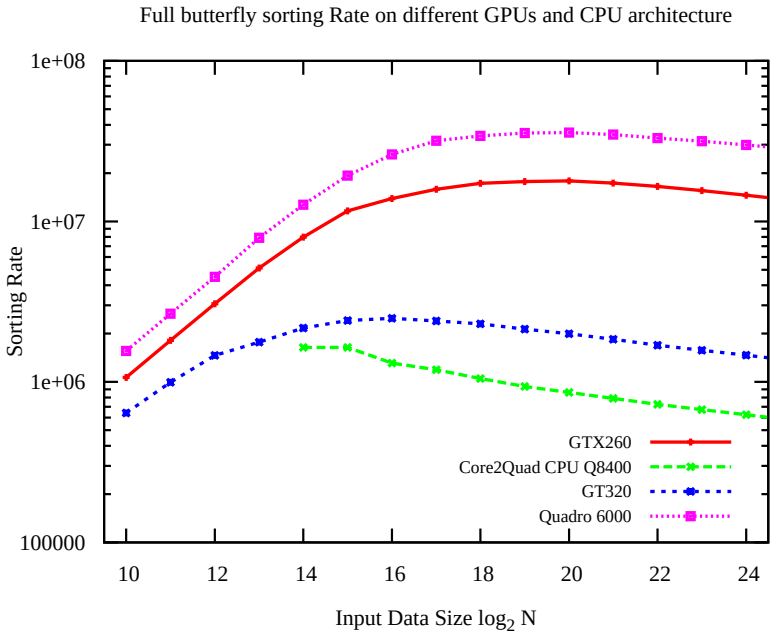

Figure 5.6. Sorting Rate Full Butterfly Sort

5.4.2.3 Speedup

Figures 5.7 and 5.8 report improvement in speedups of the butterfly sort against different sorting algorithms on different GPUs architecture where as figure 5.9 shows the speedup of serial butterfly sort against different sorting algorithms . It achieves $2x$ speedup over bitonic sort, a speedup of nearly 10^4x on Quadro-6000 over rank and odd even sort for parallel implementation and speedup of nearly 10^3x against odd-even and rank sort for serial implementation. Speedup factor increases for large queue sizes on GPUs with larger number of cores.

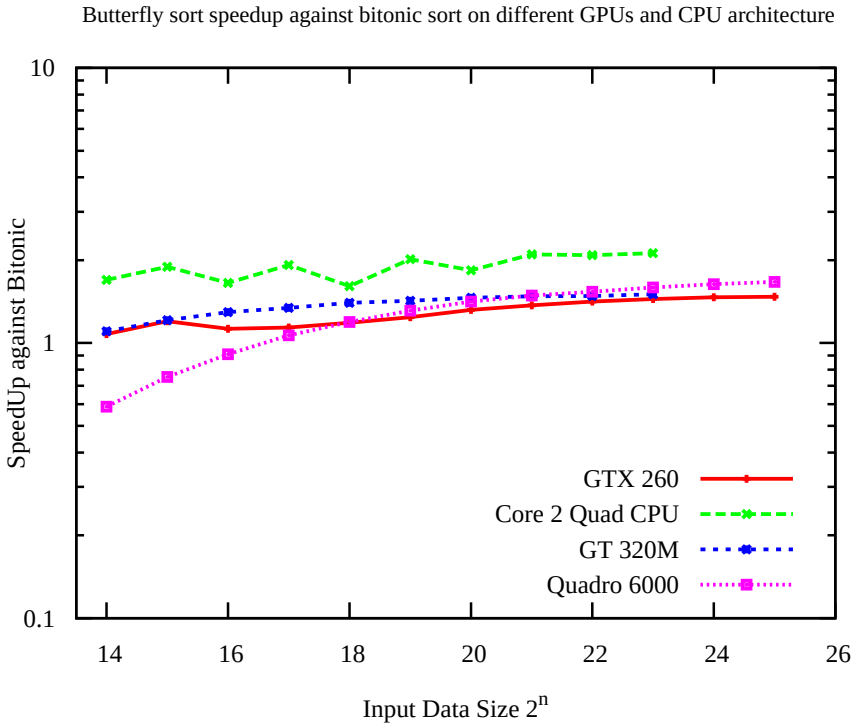

Figure 5.7. SpeedUp Full Butterfly Sort Against Bitonic Sort

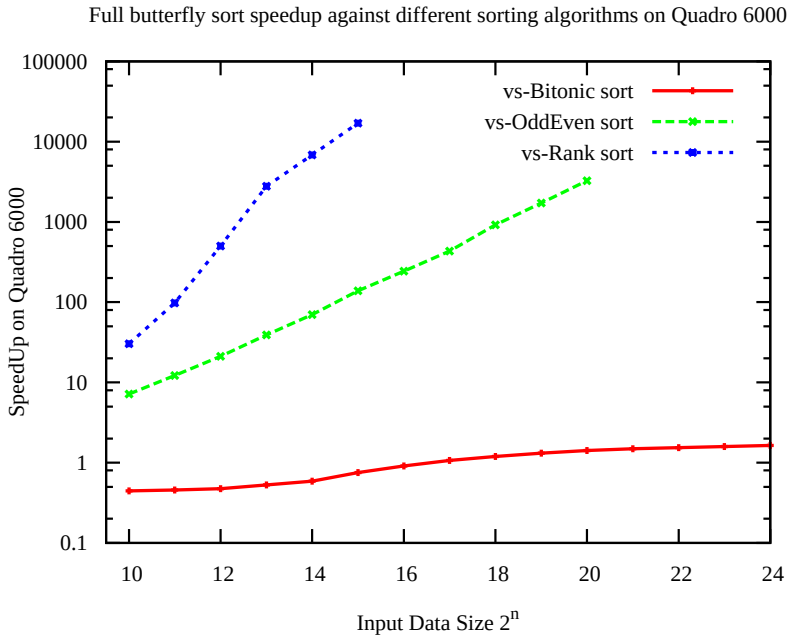

Figure 5.8. SpeedUp Full Butterfly Sort Against Different Sorting Algorithms

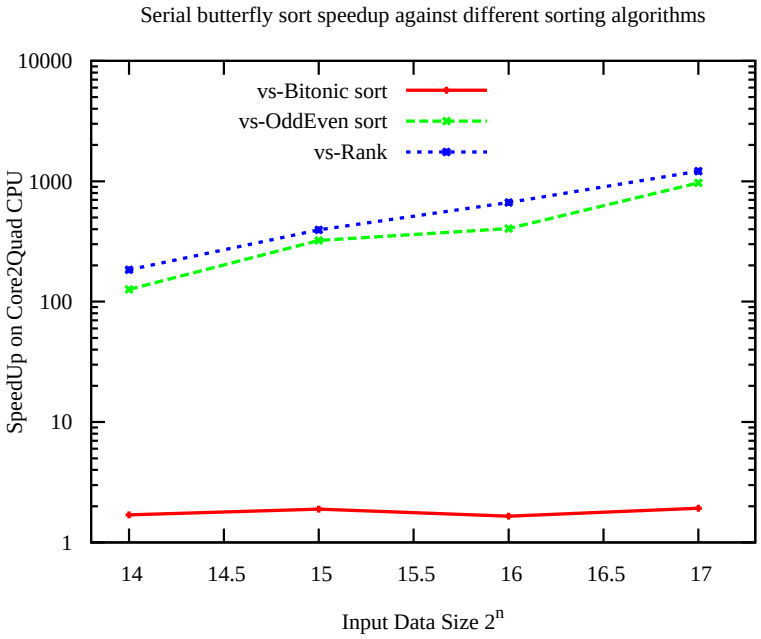

Figure 5.9. SpeedUp Serial Butterfly Sort Against Different Sorting Algorithms

5.5 Conclusion

I have tested parallel and serial implementation of novel sorting algorithms: min-max butterfly and full butterfly sorting on different GPU and CPU architectures and evaluated better performance of my algorithms in comparison to bitonic, odd-even and rank-sort in terms of sorting time, sorting rate and speedup. In future the work will be transported to multiple GPUs with optimization techniques like memory coalescing etc and uses of these algorithms for *hold—operations*. The publications related to this work are [72, 73, 74, 24]

Chapter 6

Conclusion

The graphics processing units (GPUs), which initially was designed for manipulating computer Graphics, now with the development of high level libraries and easy to use interfacing tools such as OpenCL and CUDA can be used as co-processor to speed up wide range of computation intensive applications. On the other hand in micromagnetic simulations the study of magnetization behavior at very small space and time scale requires lot of computational cost. Therefore parallelism becomes adequate for such type of simulations.

The study and observation of magnetization behavior at sub-nanosecond time-scales is crucial to a number of areas such as magnetic sensors, non volatile storage devices and magnetic nanowires etc. Since micromagnetic codes in general are suitable for parallel programming as it can be easily divided into independent parts which can run in parallel, therefore current trend for micromagnetic code concerns shifting the computationally intensive parts like the magnetostatic field calculation to GPUs. In the recent years GPUs have provided best solution to such problems both with respect to price and performance.

The current micromagnetic solvers on GPU are CUDA based and uses the general-purpose FFT library (cufft) for the computation of magnetostatic field. This limits the current GPU based magnetostatic solver to NVIDIA based hardware only. Secondly by the use of general-purpose FFT library they can not fully exploit the zero padded input data with out transposition and symmetries inherent in the field calculation, moreover on single GPU the input problem size is also an issue.

My PhD work mainly focuses on the development of highly parallel magnetostatic field solver for micromagnetic simulators on GPUs. I am using OpenCL for GPU implementation, with consideration that it is an open standard for parallel programming of heterogeneous systems for cross platform. It targets different devices such as GPUs by different vendors such as Nvidia, ATI and Intel etc, along with CPU and other processing hardware which conform to its specification. The magnetostatic field calculation is dominated by the multidimensional FFTs (Fast Fourier Transform) computation. Therefore I have developed the specialized OpenCL based 3D-FFT library for magnetostatic field calculation which made it possible to fully exploit the zero padded input data with out transposition and symmetries inherent in the field calculation. As a result the complexity of overall system reduced significantly compared to current GPU based solvers. Moreover it also provides a common interface for different vendors' GPUs. In order to fully utilize the GPUs parallel architecture my solver handles many hardware specific technicalities such as coalesced memory access, data transfer overhead between GPU and CPU, GPU global memory utilization, arithmetic computation and batch execution.

I have demonstrated the average speedup of magnetostatic field solver on different GPU architectures against widely used CPU based shared memory parallel micromagnetic solver "OOMMF" developed at NIST and against my equivalent parallel implementation respectively, running on four cpu cores. Where my GPU based implementation shows a significant speedup factor of up to 8.6 for single precision floating point accuracy with out data transfer time and 4.2 with data transfer time on high end GPU architecture that is Nvidia Quadro-6000. On the other hand the speedup against my equivalent implementation on CPU is up to 46x and 94x with and with out data transfer time respectively.

In chapter 3 I have discussed the implementation of magnetostatic field solver on single GPU and shown the speedup against OOMMF and equivalent CPU implementation for different input problem sizes. I have used different optimization techniques both with respect to our input problem and the GPU hardware architecture. Normally in micromagnetic simulations due to very fine space and time discretization the input problem sizes are very large. On the other hand on most of the GPUs have little memory resources to accommodate the very large input problem sizes. Therefore the natural next step to further increase the performance

of magnetostatic field solver on GPUs and to mitigate the limited memory problem is the use of combined resources of multiple GPUs in parallel. Therefore in the second step to further increase the level of parallelism and performance, I have developed a parallel magnetostatic field solver on multiple GPUs. Utilizing multiple GPUs avoids dealing with many of the limitations of GPUs (e.g., on-chip memory resources) by exploiting the combined resources of multiple on board GPUs.I have shown the implementation of magnetostatic field solver on multiple GPUs in parallel and the speedup against single GPU implementation. In this way we can handle very large input problem sizes by utilizing the memory resources on different GPUs along with computation speedup.

Currently my 3-D FFT library is based on Cooley Tukey radix-2 algorithm, in future I am planning to move towards higher radix or even mix radix FFT algorithms. Most importantly in order to achieve a very high performance I have to shift all the components of micromagnetic solver on GPU.

Sorting algorithms have been studied extensively since past three decades. Their uses are found in many applications including real-time systems, operating systems, and discrete event simulations. In most cases, the efficiency of an application itself depends on usage of a sorting algorithm. Lately, the usage of graphic cards for general purpose computing has again revisited sorting algorithms. I have tested parallel and serial implementation of novel sorting algorithms: min-max butterfly and full butterfly sorting on different GPU and CPU architectures and evaluated better performance of my algorithms in comparison to bitonic, odd-even and rank-sort in terms of sorting time, sorting rate and speedup.

Bibliography

[1] Paul Manuel Kalim Qureshi, Fiaz Gul Khan and Babar Nazir. A hybrid fault tolerance technique in grid computing system. *The Journal of Supercomputing*, 56:106–128, 2011. 7

[2] Babar Nazir Fiaz Gul Khan, Kalim Qureshi. Performance evaluation of fault tolerance techniques in grid computing system. *Computers and Electrical Engineering*, 36, issue 6:1110–1122, November 2010. 7

[3] Mache Creeger. Multicore cpus for the masses. *Queue, ACM New York, NY, USA*, 3(7):64–ff, 2005. 7, 29

[4] Lawrence Murray. Gpu acceleration of runge-kutta integrators. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):94–101, 2012. 7

[5] C.W. In Kyu Park; Singhal, N.; Man Hee Lee; Sungdae Cho; Kim. Design and performance evaluation of image processing algorithms on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):91–104, 2011. 7

[6] S. ; Dongarra J. Kurzak, J. ; Tomov. Autotuning gemm kernels for the fermi gpu. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):245–257, 2012. 7

[7] David Brooks. Cpus, gpus, and hybrid computing. *IEEE Micro*, 31:4–6, 2011. 7

[8] W.J. ; Khailany B. ; Garland M. ; Glasco D. Keckler, S.W. Dally. Gpus and the future of parallel computing. *IEEE Micro*, 31:7–13, 2011. 7

[9] J.D Yao Zhang, Owens. A quantitative performance analysis model for gpu architectures. In *IEEE 17th international symposium on HPCA*, 2011. 10, 11

[10] T.S. Han, T.D.; Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):78–90, 2011. 11

[11] J.; Josth R.; Herout A.; Zemcik P.; Hauta-Kasari M. Antikainen, J.; Havel.

Nonnegative tensor factorization accelerated using gpgpu. *IEEE Transactions on Parallel and Distributed Systems*, 22(7):1135–1141, 2011. 11

[12] P.; Kaeli D. Byunghyun Jang; Schaa, D.; Mistry. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, 2011. 11

[13] Mark Jason Harris. *Real-Time Cloud Simulation and Rendering*. PhD thesis, University of North Carolina at Chapel Hill, 2003. 11

[14] D. Kirk. Innovation in graphics technology. In *Talk in Canadian Undergraduate Technology Conference*, 2004. 13

[15] A.C. ; Roerdink J.B.T.M. van der Laan, W.J. ; Jalba. Accelerating wavelet lifting on graphics hardware using cuda. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):132–146, 2011. 18

[16] A. ; Hinde R.J. ; Peterson G.D. Weber, R. ; Gothandaraman. Comparing hardware accelerators in scientific applications: A case study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):58–68, 2011. 18

[17] J. Nickolls and W.J. Dally. The gpu computing era. *IEEE Micro*, 30, issue 2:56–69, 2010. 18

[18] J. Nickolls Anderson J. Hardwick S. Morton E. Phillips Y. Zang M. Garland, S. Le Grand and V. Volkov. Parallel computing experience with cuda. *IEEE Micro*, 28:13–27, 2008. 19, 46

[19] Guochun Shi Showerman M.T Arnold G.W Stone J.E Phillios J.C Wenmei Hwu Kindratenko V.V, Enos J.J. Gpu clusters for high-performance computing. In *IEEE international conference on Cluster New Orleans, LA*, 2009. 19, 46

[20] Nvidia Corporation, www.nvidia.com. *OpenCL Programming Guide for CUDA Architecture*, 2009. 25

[21] A. ; Kammerer M. ; Van Waeyenberge B. ; Dupre-L. ; De Zutter D Van de Wiele, B.; Vansteenkiste. Micromagnetic simulations on gpu, a case study: vortex core switching by high-frequency magnetic fields. *IEEE Transactions on Magnetics*, 48:2068–2072, June 2012. 29, 37

[22] Benjamin Kruger Andre Drews Claas Abert, Gunnar Selke. A fast finite-difference method for micromagnetics using the magnetic scalar potential. *IEEE Transaction on Magnetics*, 48:1105–1109, 2012. 29, 44

[23] M. J. Donahue. Parallelizing a micromagnetic program for use on multiprocessor shared memory computers. *Ieee Transactions On Magnetics*, 45:3923–3925, 2009. 29, 44, 45, 59

[24] Fiaz Gul Khan, Omar Usman Khan, Bartolomeo Montrucchio, and Paolo Giaccone. Analysis of fast parallel sorting algorithms for gpu architectures. In *Proceedings of the 2011 Frontiers of Information Technology*, pages 1173–1178. IEEE Computer Society, 2011. 30, 49, 86, 92, 97

[25] E.M. Lifshitz L.D. Landau. Theory of the dispersion of magnetic permeability in ferromagnetic bodies. *Phys. Z. Sowietunion*, 8:153–169, 1935. 31, 33

[26] W. F Brown Jr. *Micromagnetics, Interscience Tracts on Physics and Astronomy*. Willey-Interscience, New York/London, 1963. 31, 33

[27] A. Aharoni. *Introduction to the Theory of Ferromagnetism*. Oxford University Press, 2001. 31

[28] L.Dupre D.De Zutter B.VandeWiele, F.Olyslager. On the accuracy of fft based magnetostatic field evaluation schemes in micromagnetic hysteresis modeling. *Journal of Magnetism and Magnetic Materials*, 322:469–476, October, 2009. 31

[29] Massimiliano d'Aquino. *Nonlinear Magnetization Dynamics in Thin-films and Nanoparticles*. PhD thesis, Universita degli studi di Napoli Federico II, 2004. 31, 40, 41

[30] A.H Morrish. *The physical principles of magnetism*. IEEE Press, 1975. 32

[31] Nikhil S. Tambe etc. W.Merlijin Van Spengen. *Hand book of nano technology*. Springer, November 2006. 33

[32] A.E. LaBonte. Two-dimensional bloch type domains walls in ferromagnetoic films. *Appl. Physics*, 40:2450, 1969. 33

[33] J. Mallinson. On damped gyromagnetic precession. *IEEE Transactions on Magnetics*, 23:2003–2004, 2003. 33, 35, 36

[34] T.L. Gilbert. A lagrangian formulation of the gyromagnetic equation of the magnetic field. *Physical Review*, 100:1243, 1955. 33

[35] L.T. Gilbert. A phenomenological theory of damping in ferromagnetic materials. *IEEE Transactions on Magnetics*, 40:3443–3449, 2004. 33

[36] A. Hubert D. V. Berkov, K. RamstÃ¶cck. Solving micromagnetic problems. towards an optimal numerical method. *Phys. stat. sol.*, 137:207–225, 1993, Published online 2006. 36

[37] P. ; Porod W. ; Csaba G. Khan, A.A.; Lugli. Development of a highly parallelized micromagnetic simulator on graphics processors. In *14th International Workshop on Computational Electronics (IWCE)*, 2010. 36

[38] NICOLA A. SPALDIN. *Magnetic Materials Fundamentals and Applications*. Cambridge University Press New York USA, 2010. 38

[39] M.J. Donahue R.D. McMichael and D.G. Porter. Comparison of magnetostatic field calculation methods on two-dimensional square grids as applied to a micromagnetic standard problem. *Journal of Applied Physics*, 85(B):5816–5818, April 1999. 40

[40] H Neil Bertram. *Theory ofMagnetic Recording*. Cambridge University Press, 1994. 40

[41] G. Rowlands P. Rhodes. Demagnetizing energies of uniformly magnetized rectangular blocks. *Proc. Leeds Phil. Liter. Soc*, 6:191–210, 1954. 42

[42] Andrew J. NewellWyn WilliamsDavid J. Dunlop. A generalization of the demagnetizing tensor for nonuniform magnetization. *Journal of Geophysical Research*, 98(B6):9551–9555, 1993. 42, 43

[43] Schafer Rudolf Hubert, Alex. *Magnetic Domains: Analysis of Magnetic Microstructures*. Springer, 1998. 42

[44] Ragusa C. Montrucchio B. Khan O.U., Khan F. Review of parallel and distributed architectures for micromagnetic codes. *COMPEL*, In Press, 2013. 45, 69

[45] B. Van de Wiele A. Vansteenkiste. Mumax: a new high-performance micromagnetic simulation tool. *Journal of Magnetism and Magnetic Materials*, 323:2585–2591, November 2011. 45, 60

[46] M. V. Lubarda B. Livshitz R. Chang, S. Li and V. Lomakin. Fastmag: Fast micromagnetic simulator for complex magnetic structures. *Journal of Applied Physics*, 109:07D358 – 07D358–6, 2011. 45, 46

[47] Aurelio D. Torres L. Martinez E. Hernandez-Lopez M.A. Gomez J. Alejos-O. Carpentieri M. Finocchio G. Lopez-Diaz, L. and Consolo. G. Micromagnetic simulations using graphics processing units. *Journal of Physics D: Applied Physics*, 45:17, June 2012. 45, 46

[48] Elmar Westphal Attila Kakay and Riccardo Hertel. Speedup of fem micromagnetic simulations with graphical processing units. *IEEE Transaction on*

Magnetics, 46:2303–2306, 2010. 45, 46

[49] Franchin M. Bordignon G. Fischbacher, T. and H. Fangohr. A systematic approach to multiphysics extensions of finite-element-based micromagnetic simulations: Nmag. *IEEE Transaction on Magnetics*, 43:2896–2898, 2007. 45

[50] Werner Scholz. *magpar version 0.9 build 3061M.* http://www.magpar.net/static/magpar/doc/magpar.pdf, 2010. 45

[51] C. Mewes and Mewes. *M cube.* http://bama.ua.edu/ tmewes/Mcube/Mcube.shtml, October 2010. 45

[52] Yao Zhang and J.D Owens. A quantitative performance analysis model for gpu arichitectures. In *IEEE 17th International Symposium on HPCA*, 2011. 49

[53] NVIDIA, http://www.nvidia.com. *NVIDIA OpenCL Best Practices Guide*, 1.0 edition, August 2009. 54

[54] Don Porter. Mike Donahue. *OOMMF User's Guide for release 1.2 alpha 3.* http://math.nist.gov/oommf/doc/, 2002. 59

[55] F.G Khan B. Montrucchio O.U Khan, C.Ragusa. A mutual demagnetization tensor for n-body magnetic field modeling. In *12th Joint MMM/Intermag Conference Chicago, USA*, January 2013. 69

[56] V.Giovara F.G Khan O.U Khan M. Repetto C.Ragusa, B.Montrucchio and B. Xie. Implementation of 3d micromagnetic code on a parallel and distributed arichitecture. In *The fourth Italian Workshop on "The Finite Element Method Applied to Electrical and Information Engineering", Rome, Italy.* University Roma Tre, December 2010. 69

[57] V.Giovara M.Repetto F.G. Khan O.U. Khan C.Ragusa, B.Montrucchio and B.Xie. Implementation of a 3d micromagnetic code on a shared and distributed architecture. In *MAGNET 2011, 2nd Convegno Nazionale di Magnetismo, Torino, Italy*, February 2011. 69

[58] T. Schrefl D. Suess R. Dittrich-H. Forster W. Scholz, J. Fidler and V. Tsiantos. Scalable parallel micromagnetic solvers for magnetic nanostructures. *Computational Materials Science*, 28:366–383, 2003. 70

[59] J.E. ; Schulten K. Phillips, J.C. ; Stone. Adapting a message-driven parallel application to gpu-accelerated clusters. In *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*, pages 1–9. IEEE, November 2008. 71

[60] Gregorio Quintana-Orti, Francisco D. Igual, Enrique S. Quintana-Orti, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 44(4):121–130, feb 2009. 71

[61] Canqun Yang; Feng Wang ; Yunfei Du ; Juan Chen ; Jie Liu ; Huizhan Yi ; Kai Lu. Adaptive optimization for petascale heterogeneous cpu/gpu computing. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 19–28, USA, October 2010. 71

[62] Aaftab Munshi. *The OpenCL Specification*. The Khronos Group In, version 1.0, revision 48 edition, 2009. 76

[63] Andreas Bonelli, Franz Franchetti, Juergen Lorenz, Markus PÃŒschel, and ChristophW Ueberhuber. *Automatic Performance Optimization of the Discrete Fourier Transform on Distributed Memory Computers*, volume 4330 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006. 78

[64] Nvidia. *OpenCL Best Practices Guide*, April 2010. 81

[65] S.G. Akl, editor. *Parallel Sorting Algorithms*. Academic Press Inc.,U.S., 1985. 86

[66] Daniel Cederman and Philippas Tsigas. A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA)*, 14, 2010. 86

[67] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10. IEEE Computer Society, 2009. 86

[68] A. Greb and G. Zachmann. Gpu-abisort: optimal parallel sorting on stream architectures. In *20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006 Rhodes Island, Greece*. IEEE Computer Society, 2006. 86

[69] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 41–50, San Diego, California, 2003. Eurographics Association. 86

[70] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place

sorting with cuda based on bitonic sort. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, number 8 in PPAM'09, pages 403–410, Berlin, Heidelberg, 2010. Springer-Verlag. 86

[71] David Strippgen and Kai Nagel. Using common graphics hardware for multi-agent traffic simulation with cuda. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 62:1–62:8, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 86

[72] Bartolomeo Montrucchio Fiaz Gul Khan, Omar Usman Khan and Paolo Giaccone. Parallel butterfly sorting algorithm on gpus. In *The 11th IASTED International Conference on Parallel and Disributed Computing Networks, Innsbruk, Austria*, 2013. 97

[73] Carlo Ragusa Fiaz Gul Khan Omar Usman Khan Bilal Jan, Bartolomeo Montrucchio. Fast parallel sorting algorithms on gpus. *International Journal of Distributed and Parallel Systems (IJDPS)*, 3(6):107–118, 2012. 97

[74] B.Montrucchio F.G Khan, O.U Khan. A study of odd-even and rank parallel sorting algorithms for gpu. In *Innovation Information Technologies: Theory and Practice (Dresden, Germany)*, page 6, September 2010. 97